

BOOK 3: NFTS AND DIGITAL ASSETS

INTRODUCTION

Non-fungible tokens changed how we think about digital ownership. Before NFTs, digital items could be copied infinitely at zero cost. Your MP3 file was identical to everyone else's. Your in-game sword existed only at the whim of the game company. Your digital art had no scarcity, no provenance, no way to prove you owned the original.

NFTs solved this through cryptographic uniqueness. Each token has a distinct identifier. The blockchain records who owns which token. Transfers require the owner's signature. For the first time in history, digital assets could be truly owned, not just licensed.

The 2021 NFT explosion brought mainstream attention -- \$69 million for a Beeple artwork, cartoon apes selling for millions, virtual land parcels trading like Manhattan real estate. Most dismissed it as speculation. Many projects were speculation. But beneath the froth, a genuine technological shift occurred.

This book teaches you to build with NFTs. Not just create JPEGs to sell, but understand the underlying technology deeply enough to build innovative applications. Digital ownership extends far beyond art -- event tickets, credentials, memberships, game assets, music royalties, real estate deeds, identity documents. Any asset that benefits from verifiable ownership and transferability can become an NFT.

WHAT THIS BOOK COVERS

Chapter 1: NFT Fundamentals

What makes NFTs unique. How ownership works on-chain. The difference between the token and the asset it represents. Use cases beyond art. Market dynamics and valuation.

Chapter 2: ERC-721 Implementation

The standard that started it all. Interface requirements, minting functions, transfer mechanics. Building compliant contracts from scratch. Metadata structure and token URIs.

Chapter 3: ERC-1155 Multi-Token Standard

When you need both fungible and non-fungible tokens in one contract. Batch operations for efficiency. Game asset patterns. Managing multiple token types.

Chapter 4: Metadata and IPFS

The JSON that describes your NFT. Off-chain vs on-chain metadata tradeoffs. IPFS pinning and content addressing. Services like Pinata and NFT.Storage. Making metadata immutable.

Chapter 5: NFT Marketplaces Integration

Listing and selling programmatically. OpenSea Seaport protocol. Blur integration. Royalty enforcement challenges. Building your own marketplace.

Chapter 6: Decentralized Storage Deep Dive

IPFS architecture and how it works. Arweave permanent storage. Filecoin deals. Comparing storage solutions. Redundancy strategies for important assets.

Chapter 7: Advanced NFT Patterns

Dynamic NFTs that change over time. Soulbound tokens that cannot transfer. Fractionalization for expensive assets. NFT lending and collateral. Composable NFTs that combine.

Chapter 8: Building NFT Applications

Complete minting dApps. Gallery displays and collection browsers. Rarity tools and analytics. Collection management systems. Airdrop mechanics for NFT holders.

THE TECHNOLOGY STACK

NFTs combine several technologies:

Smart Contracts define the rules. ERC-721 and ERC-1155 standards

ensure interoperability. Custom logic adds minting rules, royalties, and special behaviors.

Blockchain Networks store ownership. Ethereum remains dominant for high-value NFTs. Polygon offers lower fees. Solana provides speed. Each chain has tradeoffs.

Metadata Standards describe the assets. JSON files with name, description, image, and attributes. Marketplaces parse this metadata to display NFTs correctly.

Storage Systems hold the actual files. IPFS for content-addressed storage. Arweave for permanence. Centralized servers for convenience (but with trust tradeoffs).

Indexers make data queryable. The Graph indexes NFT events. Alchemy and Moralis provide APIs. Without indexers, reading NFT data requires scanning entire blockchain history.

Marketplaces provide liquidity. OpenSea, Blur, Rarible, Magic Eden. Each with different protocols, fees, and audiences.

WHY BUILD WITH NFTS

Digital ownership enables new business models:

Creator Royalties: Artists earn on secondary sales forever. Musicians get paid when songs resell. Every future transaction compensates the original creator.

Programmable Scarcity: Limited editions with cryptographic guarantees. No counterfeits possible. Provenance tracked permanently.

Interoperable Assets: Your NFT works across any application that supports the standard. Game items can move between games. Art displays in any gallery. Tickets verify at any scanner.

Community Ownership: Token-gated access to exclusive content.

NFT holders form communities with aligned incentives. Ownership becomes membership.

Fractional Investment: Expensive assets split into affordable pieces. Art funds democratized. Real estate accessible to small investors.

PREREQUISITES

This book assumes you've completed Book 1 (Blockchain Fundamentals) or have equivalent knowledge:

- Solidity smart contract development
- Web3 libraries (ethers.js, web3.py)
- Wallet integration basics
- Transaction signing and gas concepts

If terms like ERC-20, ABI, or RPC are unfamiliar, review Book 1 first.

THE APPROACH

Each chapter follows the established pattern:

- Conceptual foundation before code
- Working examples you can deploy
- Production patterns and best practices
- Common mistakes and how to avoid them
- Integration with the broader ecosystem

By the end, you'll build complete NFT applications -- from contract deployment to frontend display to marketplace integration. The knowledge transfers to any NFT project you encounter.

Let's explore what makes non-fungible tokens work.

CHAPTER 1: NFT FUNDAMENTALS

Every NFT is a unique entry in a database. That's it. All the complexity, all the hype, all the millions of dollars -- they trace back to entries in a database that nobody controls. Understanding this fundamental simplicity is essential before diving into the

technical details.

The "non-fungible" part means each token is distinct. One Bitcoin equals any other Bitcoin -- they're fungible. But NFT #47 from a collection is different from NFT #48. They might look similar, point to similar images, have similar attributes, but the blockchain treats them as separate entities with separate owners.

This chapter establishes the conceptual foundation. What makes NFTs work. Why ownership matters. How the token relates to the asset it represents. Where NFTs add value and where they don't.

SECTION 1: WHAT MAKES NFTS UNIQUE

The technical definition: an NFT is a unique identifier stored on a blockchain with an associated owner address. That's the core. Everything else -- images, metadata, utility, community -- builds on top.

The Token vs The Asset

This distinction confuses newcomers constantly. The NFT is not the image. The NFT is a record that points to information, which might include a link to an image. When you "own an NFT," you own the blockchain entry. What that entry represents depends entirely on the contract and associated metadata.

Consider: You buy an NFT of digital art. You now own:

- The blockchain record (definitely)
- The right to sell that record (definitely)
- The image file itself (probably not -- it's usually on IPFS)
- Copyright to the image (almost never, unless explicitly granted)
- The right to use the image commercially (depends on license)

Smart contracts can encode any rules the creator wants. Some NFT projects grant full commercial rights. Others grant none. Most fall somewhere between. The NFT itself is just the ownership record -- what ownership means depends on legal agreements and social conventions outside the blockchain.

Uniqueness and Scarcity

Two related but distinct concepts:

Uniqueness means each token has a different identifier. Token ID 1 is not Token ID 2, even if they point to the same image. The blockchain distinguishes them.

Scarcity means limited quantity. A collection might have 10,000 unique tokens, or 100, or just 1. The contract controls minting and can enforce caps.

These combine to create verifiable digital scarcity. Before NFTs, claiming "only 100 digital items exist" required trust. Anyone could copy the file. With NFTs, the blockchain proves exactly how many tokens exist and who owns each one.

Provenance and History

Every NFT has a complete history recorded on-chain:

- When it was minted (created)
- Every transfer between wallets
- Every sale and price
- Who currently owns it

This provenance is valuable for authentication. An NFT claimed to be from a famous artist can be verified by checking if it was minted from the artist's known wallet. The chain of custody is public and immutable.

Traditional art authentication requires experts, documentation, and faith. NFT authentication requires only checking the blockchain.

SECTION 2: HOW OWNERSHIP WORKS

Blockchain ownership is cryptographic control. If you have the private key that controls an address, and that address is recorded as the owner of an NFT, you own it. No government, company, or

authority grants this ownership. It exists by mathematical proof.

The Ownership Model

Ownership is recorded in the smart contract's state. A mapping links token IDs to owner addresses:

```
tokenOwners[1] = 0xAbC123...
```

```
tokenOwners[2] = 0xDeF456...
```

```
tokenOwners[3] = 0xAbC123...
```

Address 0xAbC123 owns tokens 1 and 3. Address 0xDeF456 owns token 2. This mapping is the source of truth. Nothing else matters for ownership -- not what you paid, not what you think you own, not what any website says.

Transferring Ownership

Transfers update this mapping. When token 1 transfers from Alice to Bob:

Before: `tokenOwners[1] = Alice`

After: `tokenOwners[1] = Bob`

The transfer function requires Alice's signature (proving she authorized it) and updates the state. Once confirmed in a block, Bob owns the token. Alice cannot reverse this. Nobody can.

Operators and Approvals

Owners can authorize others to transfer on their behalf:

Individual approval: "Wallet X can transfer my token #5"

Operator approval: "Wallet X can transfer any of my tokens"

Marketplaces use this. When you list an NFT for sale, you approve the marketplace contract to transfer it when a buyer pays. The marketplace never takes custody -- it just has permission to execute the transfer when conditions are met.

SECTION 3: THE TOKEN-ASSET RELATIONSHIP

Here's where things get philosophically interesting. The NFT exists on-chain. The asset (image, video, music) usually exists off-chain. What connects them?

Token URI

Each NFT has a tokenURI function that returns a link to metadata. This metadata (usually JSON) contains information about the asset:

```
{
  "name": "Cool Cat #1234",
  "description": "A very cool cat",
  "image": "ipfs://QmXxx.../1234.png",
  "attributes": [
    {"trait_type": "Background", "value": "Blue"},
    {"trait_type": "Fur", "value": "Orange"},
    {"trait_type": "Eyes", "value": "Laser"}
  ]
}
```

The image field links to the actual visual. This link might use:

- IPFS (ipfs://...) -- decentralized, content-addressed
- Arweave (ar://...) -- permanent storage
- HTTP (https://...) -- centralized server

The Trust Spectrum

Different storage choices have different trust requirements:

On-chain storage: The image data is stored directly in the contract. Completely trustless. The image exists as long as the blockchain exists. But extremely expensive for anything beyond tiny images.

IPFS with pinning: Content-addressed storage where the hash guarantees integrity. If you have the hash, the content cannot change. But someone must keep the content available (pinning). If all pinners stop, the image disappears.

Arweave: Permanent storage with upfront payment for 200+ years of storage. No ongoing pinning required. Trustless permanence, but you trust Arweave's network will persist.

Centralized servers: Cheapest and fastest. But the server operator can change or remove the image at will. Your NFT could point to nothing, or to something different than what you bought.

Most NFT projects use IPFS because it balances cost, decentralization, and usability. High-value projects often use Arweave for permanence or on-chain storage for maximum trustlessness.

What You Actually Own

When you buy an NFT pointing to an IPFS image:

- You definitely own the blockchain record
- You can definitely transfer or sell that record
- The image will exist as long as someone pins it
- If unpinned, your NFT points to nothing
- You own whatever rights the project grants (varies wildly)

This is why due diligence matters. Where is the metadata stored? Who pins the IPFS content? What license comes with ownership? These questions determine whether your NFT has lasting value or becomes a pointer to nothing.

SECTION 4: USE CASES BEYOND ART

Art dominated the NFT conversation, but digital ownership applies far more broadly:

Gaming Assets

In-game items as NFTs enable:

- True ownership (the game company can't take your sword)
- Secondary markets (sell items you no longer want)
- Interoperability (use items across games that support them)

- Provenance (prove your item is from the original game)

Games like Axie Infinity, Gods Unchained, and Parallel demonstrate this model. Players own their cards, creatures, and items as NFTs tradeable on open markets.

Event Tickets

NFT tickets provide:

- Anti-counterfeiting (cryptographic authenticity)
- Programmable royalties (original seller earns on resales)
- Post-event collectibility (ticket becomes memorabilia)
- Access control (token-gated entry)

No more fake tickets. No more scalpers capturing all value. The original seller can cap resale prices or earn percentage of all secondary sales.

Memberships and Access

NFT as membership card:

- Hold token, access exclusive content
- Membership is tradeable (unlike traditional memberships)
- No central authority controlling access
- Community ownership of the membership

DAOs use this extensively. Own the token, participate in governance, access member benefits.

Credentials and Certificates

Educational credentials as NFTs:

- Verifiable degrees and certifications
- Cannot be faked
- Issuer's identity cryptographically proven
- Portable across any system

A university could issue degree NFTs. Employers verify by checking the blockchain rather than calling registrars.

Real World Assets

Physical items linked to NFTs:

- Luxury goods with NFT certificates of authenticity
- Real estate deeds as NFTs
- Vehicle titles
- Any asset benefiting from clear ownership records

This requires trusted oracles connecting physical and digital worlds, but the applications are significant.

Music and Royalties

Musicians selling NFTs that represent:

- Ownership of master recordings
- Royalty shares from streaming
- Exclusive access to unreleased content
- Direct artist-fan relationships

Platforms like Royal and Sound.xyz experiment with these models.

SECTION 5: MARKET DYNAMICS

Understanding NFT markets helps you build better applications and avoid common mistakes.

Price Discovery

NFT pricing is uniquely challenging:

- Each item is unique (no direct comparables)
- Markets are illiquid (few buyers and sellers)
- Speculation dominates (prices disconnected from utility)
- Social dynamics matter (community, status, hype)

Floor price -- the lowest listed price in a collection -- becomes a key metric. It represents the minimum cost to enter a collection. Collections with strong floors are considered healthy.

Liquidity Challenges

Most NFTs are illiquid:

- Few buyers want any specific NFT
- Wide bid-ask spreads
- Sales can take days or never happen
- Price discovery is inefficient

This differs fundamentally from fungible tokens where millions of identical units trade constantly. Building applications that improve NFT liquidity (aggregators, fractionalization, lending) addresses real problems.

Wash Trading

Fake trading to inflate volume and prices:

- Seller buys their own NFT from another wallet
- Creates appearance of demand
- Inflates collection statistics
- Misleads genuine buyers

Sophisticated platforms filter wash trades, but it remains prevalent. Volume numbers often lie.

Royalties

Creator royalties were an early NFT promise:

- Creator earns percentage on every resale
- Typically 2.5% to 10%
- Passive income from successful collections

But royalties are not enforced on-chain. Marketplaces choose whether to honor them. Blur and others made royalties optional, leading to race-to-bottom on fees. New standards attempt to enforce royalties at the contract level, but adoption is mixed.

Collection vs Individual

NFTs typically trade as collections:

- 10,000 items with shared theme
- Rarity tiers create price variation

- Floor price anchors the collection
- Rare items command premiums

Individual 1/1 art trades differently -- each piece unique, priced by artist reputation and aesthetic appeal. Collection dynamics don't apply.

SECTION 6: READING NFT DATA

Before building, you need to read existing NFT data. Here's how to query NFTs programmatically:

```
import { ethers } from 'ethers'

const ERC721_ABI = [
  'function balanceOf(address owner) view returns (uint256)',
  'function ownerOf(uint256 tokenId) view returns (address)',
  'function tokenURI(uint256 tokenId) view returns (string)',
  'function name() view returns (string)',
  'function symbol() view returns (string)',
  'function totalSupply() view returns (uint256)',
  'function tokenByIndex(uint256 index) view returns (uint256)',
  'function tokenOfOwnerByIndex(address owner, uint256 index) view returns (uint256)',
  'event Transfer(address indexed from, address indexed to, uint256 indexed tokenId)'
]

class NFTReader {
  constructor(contractAddress, provider) {
    this.contract = new ethers.Contract(contractAddress, ERC721_ABI, provider)
    this.address = contractAddress
  }

  async getCollectionInfo() {
    const [name, symbol] = await Promise.all([
      this.contract.name(),
      this.contract.symbol()
    ])
  }
}
```

```
])
```

```
let totalSupply = 0
try {
  totalSupply = await this.contract.totalSupply()
} catch (e) {
}
```

```
return { name, symbol, totalSupply: Number(totalSupply) }
}
```

```
async getOwner(tokenId) {
  try {
    return await this.contract.ownerOf(tokenId)
  } catch (e) {
    return null
  }
}
```

```
async getTokenMetadata(tokenId) {
  const uri = await this.contract.tokenURI(tokenId)
  const metadata = await this.fetchMetadata(uri)
  return metadata
}
```

```
async fetchMetadata(uri) {
  let fetchUrl = uri
```

```
  if (uri.startsWith('ipfs://')) {
    const hash = uri.replace('ipfs://', '')
    fetchUrl = `https://ipfs.io/ipfs/${hash}`
  }
```

```
  if (uri.startsWith('ar://')) {
    const hash = uri.replace('ar://', '')
    fetchUrl = `https://arweave.net/${hash}`
  }
```

```
  if (uri.startsWith('data:application/json;base64,')) {
    const base64 = uri.replace('data:application/json;base64,', '')
```

```
const json = atob(base64)
return JSON.parse(json)
}
```

```
const response = await fetch(fetchUrl)
return response.json()
}
```

```
async getTokensOwnedBy(ownerAddress) {
const balance = await this.contract.balanceOf(ownerAddress)
const tokens = []
```

```
for (let i = 0; i < balance; i++) {
  try {
    const tokenId = await
this.contract.tokenOfOwnerByIndex(ownerAddress, i)
    tokens.push(Number(tokenId))
  } catch (e) {
    break
  }
}
```

```
return tokens
}
```

```
async getTransferHistory(tokenId, fromBlock = 0) {
const filter = this.contract.filters.Transfer(null, null, tokenId)
const events = await this.contract.queryFilter(filter, fromBlock)
```

```
return events.map(event => ({
  from: event.args.from,
  to: event.args.to,
  tokenId: Number(event.args.tokenId),
  blockNumber: event.blockNumber,
  transactionHash: event.transactionHash
}))
}
```

```
async getAllTransfers(fromBlock = 0, toBlock = 'latest') {
const filter = this.contract.filters.Transfer()
```

```
const events = await this.contract.queryFilter(filter, fromBlock,
toBlock)
```

```
return events.map(event => ({
  from: event.args.from,
  to: event.args.to,
  tokenId: Number(event.args.tokenId),
  blockNumber: event.blockNumber,
  transactionHash: event.transactionHash
}))
}
}
```

```
async function analyzeCollection(contractAddress, provider) {
  const reader = new NFTReader(contractAddress, provider)
```

```
  const info = await reader.getCollectionInfo()
  console.log(`Collection: ${info.name} (${info.symbol})`)
  console.log(`Total Supply: ${info.totalSupply}`)
```

```
  const sampleTokenId = 1
  const owner = await reader.getOwner(sampleTokenId)
  console.log(`Token #${sampleTokenId} owned by: ${owner}`)
```

```
  const metadata = await reader.getTokenMetadata(sampleTokenId)
  console.log(`Metadata:`, metadata)
```

```
  const history = await reader.getTransferHistory(sampleTokenId)
  console.log(`Transfer history:`, history)
}
```

SECTION 7: ANALYZING NFT COLLECTIONS

Understanding collections requires analyzing holder distribution, rarity, and trading patterns:

```
class CollectionAnalyzer {
  constructor(nftReader) {
    this.reader = nftReader
```



```
}
```

```
async getHolderDistribution(totalSupply) {  
  const holders = {}
```

```
  for (let tokenId = 1; tokenId <= totalSupply; tokenId + +) {  
    const owner = await this.reader.getOwner(tokenId)  
    if (owner) {  
      holders[owner] = (holders[owner] || 0) + 1  
    }  
  }  
}
```

```
const distribution = Object.entries(holders)  
  .map(([address, count]) => ({ address, count })))  
  .sort((a, b) => b.count - a.count)
```

```
return {  
  uniqueHolders: distribution.length,  
  distribution: distribution,  
  topHolders: distribution.slice(0, 10),  
  averagePerHolder: totalSupply / distribution.length  
}  
}
```

```
async analyzeRarity(totalSupply) {  
  const attributes = {}  
  const tokens = []
```

```
  for (let tokenId = 1; tokenId <= Math.min(totalSupply, 100);  
    tokenId + +) {  
    try {  
      const metadata = await this.reader.getTokenMetadata(tokenId)
```

```
      if (metadata.attributes) {  
        for (const attr of metadata.attributes) {  
          const key = attr.trait_type  
          const value = attr.value
```

```
        if (!attributes[key]) {  
          attributes[key] = {}
```

```

}
attributes[key][value] = (attributes[key][value] || 0) + 1
}
}

```

```

tokens.push({ tokenId, metadata })
} catch (e) {
  continue
}
}

```

```

const rarityScores = tokens.map(token => {
  let score = 0
  const traits = token.metadata.attributes || []

```

```

  for (const attr of traits) {
    const traitCount = attributes[attr.trait_type][attr.value]
    const traitRarity = 1 / (traitCount / tokens.length)
    score += traitRarity
  }

```

```

  return { tokenId: token.tokenId, rarityScore: score }
})

```

```

return {
  attributes,
  rarityScores: rarityScores.sort((a, b) => b.rarityScore -
a.rarityScore)
}
}

```

```

calculateConcentration(distribution) {
  const total = distribution.reduce((sum, h) => sum + h.count, 0)

```

```

  const top10Count = distribution.slice(0, 10).reduce((sum, h) =>
sum + h.count, 0)
  const top10Percent = (top10Count / total * 100).toFixed(2)

```

```

  let cumulative = 0
  let giniSum = 0

```

```

const n = distribution.length

for (let i = 0; i < n; i++) {
  cumulative += distribution[n - 1 - i].count
  giniSum += cumulative
}

const gini = (2 * giniSum) / (n * total) - (n + 1) / n

return {
  top10Concentration: top10Percent,
  giniCoefficient: gini.toFixed(4)
}
}
}

```

SECTION 8: NFT VALUATION CONCEPTS

Valuing NFTs is more art than science, but certain metrics help:

```

class NFTValuation {
  constructor() {
    this.weights = {
      rarity: 0.3,
      utility: 0.2,
      community: 0.2,
      artistReputation: 0.15,
      historicalSales: 0.15
    }
  }

  calculateRarityScore(attributes, collectionAttributes) {
    let score = 0

    for (const attr of attributes) {
      const traitType = attr.trait_type
      const value = attr.value

      if (collectionAttributes[traitType] &&

```

```

collectionAttributes[traitType][value]) {
  const frequency = collectionAttributes[traitType]
[value].frequency
  score += (1 - frequency) * 100
}
}

```

```

return score
}

```

```

estimateFloorMultiple(rarityScore, averageRarityScore) {
  const rarityRatio = rarityScore / averageRarityScore

```

```

  if (rarityRatio > 3) return 10
  if (rarityRatio > 2) return 5
  if (rarityRatio > 1.5) return 2.5
  if (rarityRatio > 1.2) return 1.5
  return 1
}

```

```

analyzeComparables(recentSales, targetAttributes) {
  const similarSales = recentSales.filter(sale => {
    const matchingTraits = sale.attributes.filter(attr =>
targetAttributes.some(t =>
t.trait_type === attr.trait_type && t.value === attr.value
)
)
    return matchingTraits.length >= targetAttributes.length * 0.5
  })

```

```

  if (similarSales.length === 0) return null

```

```

  const prices = similarSales.map(s => s.price)
  const avgPrice = prices.reduce((a, b) => a + b, 0) /
prices.length
  const medianPrice = prices.sort((a, b) => a - b)
[Math.floor(prices.length / 2)]

```

```

  return {
    comparableCount: similarSales.length,

```

```
averagePrice: avgPrice,  
medianPrice: medianPrice,  
priceRange: { min: Math.min(...prices), max: Math.max(...prices) }  
}  
}
```

```
assessLiquidity(collection) {  
  const volumeScore =  
this.normalizeVolume(collection.dailyVolume)  
  const salesScore = this.normalizeSales(collection.dailySales)  
  const listingRatio = collection.listed / collection.totalSupply  
  
  return {  
    volumeScore,  
    salesScore,  
    listingRatio,  
    liquidityRating: (volumeScore + salesScore) / 2 > 0.5 ? 'high' :  
'low'  
  }  
}
```

```
normalizeVolume(volume) {  
  return Math.min(volume / 100, 1)  
}
```

```
normalizeSales(sales) {  
  return Math.min(sales / 50, 1)  
}  
}
```

SECTION 9: COMMON MISTAKES AND MISCONCEPTIONS

Understanding what NFTs are NOT prevents costly errors:

Misconception: Owning the NFT means owning the copyright

Reality: NFT ownership and copyright are separate. Buying an NFT gives you the token. Copyright remains with the creator unless explicitly transferred. Most NFT licenses grant limited display

rights, not commercial rights. Always check the license.

Misconception: The image is stored on the blockchain

Reality: Images are almost never on-chain (too expensive). The token points to metadata which points to an image hosted elsewhere. If that hosting fails, your NFT points to nothing. Check where assets are stored before buying.

Misconception: NFTs are immutable

Reality: The token ID and ownership history are immutable. But:

- Metadata can be changeable if the contract allows
- Images can be swapped if using HTTP URLs
- Contracts can have admin functions that modify behavior

Only buy NFTs with verifiable immutable metadata (IPFS, Arweave, or on-chain).

Misconception: High floor price means safe investment

Reality: NFT floors can collapse quickly. Illiquidity means you might not find buyers at any price. Most collections go to zero. Treat NFT purchases as high-risk speculation unless you're buying for utility or collecting.

Misconception: Royalties are guaranteed

Reality: Royalties are suggestions, not requirements. Marketplaces choose whether to honor them. Many don't. New standards attempt enforcement, but adoption is inconsistent.

SECTION 10: BUILDING YOUR MENTAL MODEL

Before writing NFT code, internalize these concepts:

The Token is the Product

The NFT itself -- the blockchain record -- is what you're creating

and selling. The image, the metadata, the utility are supporting materials. Focus on the on-chain mechanics first.

Ownership is Absolute

Whoever controls the owner address controls the NFT. No appeals, no customer service, no reversals. Build interfaces that make this clear to users.

Storage is a Spectrum

Choose storage based on permanence requirements:

- On-chain: Maximum permanence, maximum cost
- Arweave: High permanence, moderate cost
- IPFS: Good permanence if pinned, low cost
- HTTP: No guarantees, lowest cost

Match storage to asset value and user expectations.

Standards Enable Ecosystems

ERC-721 and ERC-1155 work because everyone follows the same interface. Wallets, marketplaces, and tools can support any compliant NFT. Custom interfaces break compatibility.

Metadata Matters

How you structure metadata determines how NFTs display across platforms. Follow conventions. Test on major marketplaces before launch.

Gas Costs Shape Design

Minting costs money. Batch operations save gas. Lazy minting defers costs to buyers. Design with gas awareness.

The next chapter implements ERC-721 from scratch, translating these concepts into working code. You'll understand every function, every event, every design choice that makes NFTs work.

CHAPTER 2: ERC-721 IMPLEMENTATION

ERC-721 is the standard that made NFTs possible. Published in January 2018, it defined how non-fungible tokens should work on Ethereum. Every major NFT collection -- CryptoPunks (retrofitted), Bored Apes, Art Blocks, and thousands more -- implements this standard.

Understanding ERC-721 deeply means understanding NFTs deeply. This chapter builds the standard from first principles. Every function, every event, every design decision explained. By the end, you'll implement production-ready NFT contracts and understand exactly why they work.

SECTION 1: THE STANDARD INTERFACE

ERC-721 defines required functions and events. Any contract implementing these correctly can call itself ERC-721 compliant. Wallets, marketplaces, and tools rely on this compliance.

Required Functions

```
interface IERC721 {  
    function balanceOf(address owner) external view returns (uint256  
balance);
```

```
    function ownerOf(uint256 tokenId) external view returns (address  
owner);
```

```
    function safeTransferFrom(  
address from,  
address to,  
uint256 tokenId,  
bytes calldata data  
) external;
```

```
    function safeTransferFrom(  
address from,  
address to,  
uint256 tokenId
```



```
) external;
```

```
function transferFrom(  
address from,  
address to,  
uint256 tokenId  
) external;
```

```
function approve(address to, uint256 tokenId) external;
```

```
function setApprovalForAll(address operator, bool approved)  
external;
```

```
function getApproved(uint256 tokenId) external view returns  
(address operator);
```

```
function isApprovedForAll(  
address owner,  
address operator  
) external view returns (bool);  
}
```

Required Events

```
event Transfer(address indexed from, address indexed to, uint256  
indexed tokenId);
```

```
event Approval(address indexed owner, address indexed approved,  
uint256 indexed tokenId);
```

```
event ApprovalForAll(address indexed owner, address indexed  
operator, bool approved);
```

What Each Function Does

balanceOf: Returns how many NFTs an address owns. Used to check if someone holds any tokens from a collection.

ownerOf: Returns the owner of a specific token ID. The fundamental ownership query.

transferFrom: Moves a token from one address to another. Requires

caller to be owner or approved.

safeTransferFrom: Same as transferFrom but checks if the recipient can receive NFTs (important for contracts).

approve: Authorizes a specific address to transfer a specific token. One approval per token.

setApprovalForAll: Authorizes an address to transfer ALL tokens owned by the caller. Used for marketplace listings.

getApproved: Returns the approved address for a specific token.

isApprovedForAll: Checks if an operator is approved for all tokens of an owner.

SECTION 2: IMPLEMENTING FROM SCRATCH

Let's build a complete ERC-721 implementation:

```
contract ERC721 {
    string public name;
    string public symbol;

    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;
    mapping(uint256 => address) private _tokenApprovals;
    mapping(address => mapping(address => bool)) private
        _operatorApprovals;

    event Transfer(address indexed from, address indexed to, uint256
        indexed tokenId);
    event Approval(address indexed owner, address indexed approved,
        uint256 indexed tokenId);
    event ApprovalForAll(address indexed owner, address indexed
        operator, bool approved);

    constructor(string memory tokenName, string memory
        tokenSymbol) {
```

```
name = tokenName;  
symbol = tokenSymbol;  
}
```

```
function balanceOf(address owner) public view returns (uint256) {  
    require(owner != address(0));  
    return _balances[owner];  
}
```

```
function ownerOf(uint256 tokenId) public view returns (address) {  
    address owner = _owners[tokenId];  
    require(owner != address(0));  
    return owner;  
}
```

```
function approve(address to, uint256 tokenId) public {  
    address owner = ownerOf(tokenId);  
    require(to != owner);  
    require(msg.sender == owner || isApprovedForAll(owner,  
msg.sender));
```

```
    _tokenApprovals[tokenId] = to;  
    emit Approval(owner, to, tokenId);  
}
```

```
function getApproved(uint256 tokenId) public view returns  
(address) {  
    require(_owners[tokenId] != address(0));  
    return _tokenApprovals[tokenId];  
}
```

```
function setApprovalForAll(address operator, bool approved)  
public {  
    require(operator != msg.sender);  
    _operatorApprovals[msg.sender][operator] = approved;  
    emit ApprovalForAll(msg.sender, operator, approved);  
}
```

```
function isApprovedForAll(address owner, address operator) public  
view returns (bool) {
```

```
return _operatorApprovals[owner][operator];  
}
```

```
function transferFrom(address from, address to, uint256 tokenId)  
public {  
    require(_isApprovedOrOwner(msg.sender, tokenId));  
    _transfer(from, to, tokenId);  
}
```

```
function safeTransferFrom(address from, address to, uint256  
tokenId) public {  
    safeTransferFrom(from, to, tokenId, "");  
}
```

```
function safeTransferFrom(  
    address from,  
    address to,  
    uint256 tokenId,  
    bytes memory data  
) public {  
    require(_isApprovedOrOwner(msg.sender, tokenId));  
    _safeTransfer(from, to, tokenId, data);  
}
```

```
function _safeTransfer(  
    address from,  
    address to,  
    uint256 tokenId,  
    bytes memory data  
) internal {  
    _transfer(from, to, tokenId);  
    require(_checkOnERC721Received(from, to, tokenId, data));  
}
```

```
function _transfer(address from, address to, uint256 tokenId)  
internal {  
    require(ownerOf(tokenId) == from);  
    require(to != address(0));  
  
    _tokenApprovals[tokenId] = address(0);
```

```
emit Approval(from, address(0), tokenId);
```

```
_balances[from] -= 1;
```

```
_balances[to] += 1;
```

```
_owners[tokenId] = to;
```

```
emit Transfer(from, to, tokenId);
```

```
}
```

```
function _mint(address to, uint256 tokenId) internal {
```

```
require(to != address(0));
```

```
require(_owners[tokenId] == address(0));
```

```
_balances[to] += 1;
```

```
_owners[tokenId] = to;
```

```
emit Transfer(address(0), to, tokenId);
```

```
}
```

```
function _burn(uint256 tokenId) internal {
```

```
address owner = ownerOf(tokenId);
```

```
_tokenApprovals[tokenId] = address(0);
```

```
_balances[owner] -= 1;
```

```
delete _owners[tokenId];
```

```
emit Transfer(owner, address(0), tokenId);
```

```
}
```

```
function _isApprovedOrOwner(address spender, uint256 tokenId)
```

```
internal view returns (bool) {
```

```
address owner = ownerOf(tokenId);
```

```
return (
```

```
spender == owner ||
```

```
getApproved(tokenId) == spender ||
```

```
isApprovedForAll(owner, spender)
```

```
);
```

```
}
```

```

function _checkOnERC721Received(
    address from,
    address to,
    uint256 tokenId,
    bytes memory data
) private returns (bool) {
    if (to.code.length > 0) {
        try IERC721Receiver(to).onERC721Received(msg.sender, from,
            tokenId, data) returns (bytes4 retval) {
            return retval == IERC721Receiver.onERC721Received.selector;
        } catch (bytes memory reason) {
            if (reason.length == 0) {
                revert("Transfer to non-receiver");
            } else {
                assembly {
                    revert(add(32, reason), mload(reason))
                }
            }
        }
    } else {
        return true;
    }
}

```

```

function supportsInterface(bytes4 interfaceId) public pure returns
(bool) {
    return
        interfaceId == 0x80ac58cd ||
        interfaceId == 0x5b5e139f ||
        interfaceId == 0x01ffc9a7;
}

```

```

interface IERC721Receiver {
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4);
}

```

```
}
```

Understanding the Storage

Four mappings store all state:

`_owners` maps token IDs to owner addresses. This is the ownership record.

`_balances` tracks how many tokens each address owns. Maintained alongside `_owners` for efficient balanceOf queries.

`_tokenApprovals` maps token IDs to approved addresses. Each token can have one approved address.

`_operatorApprovals` maps owner addresses to operator addresses to approval status. Allows bulk approval for all tokens.

SECTION 3: METADATA EXTENSION

The base ERC-721 doesn't include metadata. The ERC721Metadata extension adds it:

```
interface IERC721Metadata {  
    function name() external view returns (string memory);  
    function symbol() external view returns (string memory);  
    function tokenURI(uint256 tokenId) external view returns (string  
memory);  
}
```

Implementing tokenURI requires deciding how to generate URIs:

```
contract ERC721WithMetadata is ERC721 {  
    string private _baseTokenURI;  
  
    constructor(  
        string memory tokenName,  
        string memory tokenSymbol,  
        string memory baseURI
```

```
) ERC721(tokenName, tokenSymbol) {
_baseTokenURI = baseURI;
}
```

```
function tokenURI(uint256 tokenId) public view returns (string
memory) {
require(_owners[tokenId] != address(0));
return string(abi.encodePacked(_baseTokenURI, toString(tokenId),
".json"));
}
```

```
function setBaseURI(string memory newBaseURI) external {
_baseTokenURI = newBaseURI;
}
```

```
function toString(uint256 value) internal pure returns (string
memory) {
if (value == 0) {
return "0";
}
uint256 temp = value;
uint256 digits;
while (temp != 0) {
digits++;
temp /= 10;
}
bytes memory buffer = new bytes(digits);
while (value != 0) {
digits--;
buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
value /= 10;
}
return string(buffer);
}
}
```

Common tokenURI patterns:

Base URI + Token ID: ipfs://QmXxx.../1.json, ipfs://
QmXxx.../2.json

Each token gets sequential JSON file.

Base URI + Token ID + Extension: <https://api.example.com/token/1.json>

Centralized API serves metadata.

On-chain generation: Data URI with base64 encoded JSON.

Most trustless but limited by gas costs.

SECTION 4: ENUMERATION EXTENSION

ERC721Enumerable adds functions to list all tokens and tokens by owner:

```
interface IERC721Enumerable {  
    function totalSupply() external view returns (uint256);  
    function tokenByIndex(uint256 index) external view returns  
(uint256);  
    function tokenOfOwnerByIndex(address owner, uint256 index)  
external view returns (uint256);  
}
```

Implementation requires additional storage:

```
contract ERC721Enumerable is ERC721 {  
    uint256[] private _allTokens;  
    mapping(uint256 => uint256) private _allTokensIndex;  
    mapping(address => uint256[]) private _ownedTokens;  
    mapping(uint256 => uint256) private _ownedTokensIndex;  
  
    constructor(  
        string memory tokenName,  
        string memory tokenSymbol  
    ) ERC721(tokenName, tokenSymbol) {}  
  
    function totalSupply() public view returns (uint256) {  
        return _allTokens.length;  
    }  
}
```

```
function tokenByIndex(uint256 index) public view returns  
(uint256) {  
    require(index < _allTokens.length);  
    return _allTokens[index];  
}
```

```
function tokenOfOwnerByIndex(address owner, uint256 index)  
public view returns (uint256) {  
    require(index < _balances[owner]);  
    return _ownedTokens[owner][index];  
}
```

```
function _mint(address to, uint256 tokenId) internal override {  
    super._mint(to, tokenId);  
    _addTokenToAllTokensEnumeration(tokenId);  
    _addTokenToOwnerEnumeration(to, tokenId);  
}
```

```
function _transfer(address from, address to, uint256 tokenId)  
internal override {  
    super._transfer(from, to, tokenId);  
    _removeTokenFromOwnerEnumeration(from, tokenId);  
    _addTokenToOwnerEnumeration(to, tokenId);  
}
```

```
function _burn(uint256 tokenId) internal override {  
    address owner = ownerOf(tokenId);  
    super._burn(tokenId);  
    _removeTokenFromOwnerEnumeration(owner, tokenId);  
    _removeTokenFromAllTokensEnumeration(tokenId);  
}
```

```
function _addTokenToAllTokensEnumeration(uint256 tokenId)  
private {  
    _allTokensIndex[tokenId] = _allTokens.length;  
    _allTokens.push(tokenId);  
}
```

```
function _addTokenToOwnerEnumeration(address to, uint256  
tokenId) private {
```

```

_ownedTokensIndex[tokenId] = _ownedTokens[to].length;
_ownedTokens[to].push(tokenId);
}

function _removeTokenFromOwnerEnumeration(address from,
uint256 tokenId) private {
    uint256 lastTokenIndex = _ownedTokens[from].length - 1;
    uint256 tokenIndex = _ownedTokensIndex[tokenId];

    if (tokenIndex != lastTokenIndex) {
        uint256 lastTokenId = _ownedTokens[from][lastTokenIndex];
        _ownedTokens[from][tokenIndex] = lastTokenId;
        _ownedTokensIndex[lastTokenId] = tokenIndex;
    }

    _ownedTokens[from].pop();
    delete _ownedTokensIndex[tokenId];
}

```

```

function _removeTokenFromAllTokensEnumeration(uint256
tokenId) private {
    uint256 lastTokenIndex = _allTokens.length - 1;
    uint256 tokenIndex = _allTokensIndex[tokenId];
    uint256 lastTokenId = _allTokens[lastTokenIndex];

    _allTokens[tokenIndex] = lastTokenId;
    _allTokensIndex[lastTokenId] = tokenIndex;

    _allTokens.pop();
    delete _allTokensIndex[tokenId];
}
}

```

Trade-off: Enumeration adds significant gas costs to mints, transfers, and burns. Many modern collections skip it to save gas, relying on indexers to track tokens instead.

SECTION 5: BUILDING A COMPLETE NFT CONTRACT

Combining everything into a production-ready contract:

```
contract MyNFTCollection is ERC721 {
    address public owner;
    uint256 public totalSupply;
    uint256 public maxSupply;
    uint256 public mintPrice;

    string private baseTokenURI;
    bool public mintingEnabled;
    bool public revealed;
    string private hiddenMetadataURI;

    mapping(address => uint256) public mintedPerWallet;
    uint256 public maxPerWallet;

    constructor(
        string memory tokenName,
        string memory tokenSymbol,
        uint256 supply,
        uint256 price,
        uint256 walletLimit,
        string memory hiddenURI
    ) ERC721(tokenName, tokenSymbol) {
        owner = msg.sender;
        maxSupply = supply;
        mintPrice = price;
        maxPerWallet = walletLimit;
        hiddenMetadataURI = hiddenURI;
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function mint(uint256 quantity) external payable {
        require(mintingEnabled);
        require(totalSupply + quantity <= maxSupply);
        require(mintedPerWallet[msg.sender] + quantity <=
```

```

maxPerWallet);
require(msg.value >= mintPrice * quantity);

for (uint256 i = 0; i < quantity; i++) {
    totalSupply++;
    _mint(msg.sender, totalSupply);
}

mintedPerWallet[msg.sender] += quantity;
}

function ownerMint(address to, uint256 quantity) external
onlyOwner {
    require(totalSupply + quantity <= maxSupply);

    for (uint256 i = 0; i < quantity; i++) {
        totalSupply++;
        _mint(to, totalSupply);
    }
}

function tokenURI(uint256 tokenId) public view returns (string
memory) {
    require(_owners[tokenId] != address(0));

    if (!revealed) {
        return hiddenMetadataURI;
    }

    return string(abi.encodePacked(baseTokenURI, toString(tokenId),
".json"));
}

function setBaseURI(string memory newBaseURI) external
onlyOwner {
    baseTokenURI = newBaseURI;
}

function setMintPrice(uint256 newPrice) external onlyOwner {
    mintPrice = newPrice;
}

```

```
}
```

```
function setMaxPerWallet(uint256 newMax) external onlyOwner {  
    maxPerWallet = newMax;  
}
```

```
function enableMinting(bool enabled) external onlyOwner {  
    mintingEnabled = enabled;  
}
```

```
function reveal(string memory newBaseURI) external onlyOwner {  
    revealed = true;  
    baseTokenURI = newBaseURI;  
}
```

```
function withdraw() external onlyOwner {  
    uint256 balance = address(this).balance;  
    payable(owner).transfer(balance);  
}
```

```
function toString(uint256 value) internal pure returns (string  
memory) {  
    if (value == 0) return "0";  
    uint256 temp = value;  
    uint256 digits;  
    while (temp != 0) {  
        digits++;  
        temp /= 10;  
    }  
    bytes memory buffer = new bytes(digits);  
    while (value != 0) {  
        digits--;  
        buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));  
        value /= 10;  
    }  
    return string(buffer);  
}  
}
```

Features included:

- Public minting with price
- Owner reserve minting
- Per-wallet limits
- Hidden metadata until reveal
- Configurable parameters
- Withdrawal function

SECTION 6: WHITELIST AND ALLOWLIST PATTERNS

Most launches include whitelists for early or guaranteed access:

Mapping-Based Whitelist

```
contract WhitelistNFT is ERC721 {
    mapping(address => bool) public whitelist;
    mapping(address => uint256) public whitelistMinted;

    uint256 public whitelistPrice;
    uint256 public whitelistMaxPerWallet;
    bool public whitelistSaleActive;

    address public owner;
    uint256 public totalSupply;
    uint256 public maxSupply;

    constructor(
        string memory tokenName,
        string memory tokenSymbol,
        uint256 supply
    ) ERC721(tokenName, tokenSymbol) {
        owner = msg.sender;
        maxSupply = supply;
    }

    function addToWhitelist(address[] calldata addresses) external {
        require(msg.sender == owner);
        for (uint256 i = 0; i < addresses.length; i++) {
            whitelist[addresses[i]] = true;
        }
    }
}
```

```
}
```

```
function removeFromWhitelist(address[] calldata addresses)
external {
    require(msg.sender == owner);
    for (uint256 i = 0; i < addresses.length; i++) {
        whitelist[addresses[i]] = false;
    }
}
```

```
function whitelistMint(uint256 quantity) external payable {
    require(whitelistSaleActive);
    require(whitelist[msg.sender]);
    require(whitelistMinted[msg.sender] + quantity <=
whitelistMaxPerWallet);
    require(totalSupply + quantity <= maxSupply);
    require(msg.value >= whitelistPrice * quantity);

    for (uint256 i = 0; i < quantity; i++) {
        totalSupply++;
        _mint(msg.sender, totalSupply);
    }
}
```

```
whitelistMinted[msg.sender] += quantity;
}
}
```

Problem: Adding thousands of addresses costs significant gas.

Merkle Tree Whitelist

More gas-efficient for large whitelists:

```
contract MerkleWhitelistNFT is ERC721 {
    bytes32 public merkleRoot;
    mapping(address => bool) public claimed;

    address public owner;
    uint256 public totalSupply;
    uint256 public maxSupply;
```



```
uint256 public mintPrice;  
bool public saleActive;
```

```
constructor(  
string memory tokenName,  
string memory tokenSymbol,  
uint256 supply,  
bytes32 root  
) ERC721(tokenName, tokenSymbol) {  
owner = msg.sender;  
maxSupply = supply;  
merkleRoot = root;  
}
```

```
function whitelistMint(  
uint256 quantity,  
uint256 maxQuantity,  
bytes32[] calldata merkleProof  
) external payable {  
require(saleActive);  
require(!claimed[msg.sender]);  
require(quantity <= maxQuantity);  
require(totalSupply + quantity <= maxSupply);  
require(msg.value >= mintPrice * quantity);
```

```
bytes32 leaf = keccak256(abi.encodePacked(msg.sender,  
maxQuantity));  
require(verify(merkleProof, merkleRoot, leaf));
```

```
claimed[msg.sender] = true;
```

```
for (uint256 i = 0; i < quantity; i++) {  
totalSupply++;  
_mint(msg.sender, totalSupply);  
}  
}
```

```
function verify(  
bytes32[] memory proof,  
bytes32 root,
```

```

bytes32 leaf
) internal pure returns (bool) {
bytes32 computedHash = leaf;

for (uint256 i = 0; i < proof.length; i + +) {
bytes32 proofElement = proof[i];

if (computedHash <= proofElement) {
computedHash = keccak256(abi.encodePacked(computedHash,
proofElement));
} else {
computedHash = keccak256(abi.encodePacked(proofElement,
computedHash));
}
}

return computedHash == root;
}

function setMerkleRoot(bytes32 newRoot) external {
require(msg.sender == owner);
merkleRoot = newRoot;
}
}

```

Generating the merkle tree off-chain:

```

import { StandardMerkleTree } from '@openzeppelin/merkle-tree'

function generateWhitelistTree(whitelist) {
const values = whitelist.map(entry => [entry.address,
entry.maxQuantity.toString()])

const tree = StandardMerkleTree.of(values, ['address', 'uint256'])

console.log('Merkle Root:', tree.root)

const proofs = {}
for (const [i, v] of tree.entries()) {
proofs[v[0]] = {

```

```

maxQuantity: v[1],
proof: tree.getProof(i)
}
}

return { root: tree.root, proofs }
}

```

```

const whitelist = [
  { address: '0x1111111111111111111111111111111111111111111111111111111111111111',
    maxQuantity: 2 },
  { address: '0x2222222222222222222222222222222222222222222222222222222222222222',
    maxQuantity: 1 },
  { address: '0x3333333333333333333333333333333333333333333333333333333333333333',
    maxQuantity: 3 }
]

```

```

const { root, proofs } = generateWhitelistTree(whitelist)

```

SECTION 7: GAS OPTIMIZATION TECHNIQUES

NFT minting should be as cheap as possible:

ERC721A Pattern

ERC721A (by Azuki) optimizes batch minting:

```

contract ERC721A {
  string public name;
  string public symbol;

  uint256 private currentIndex;

  struct TokenOwnership {
    address addr;
    uint64 startTimestamp;
  }

  mapping(uint256 => TokenOwnership) private _ownerships;

```

```
mapping(address => uint256) private _balances;
mapping(uint256 => address) private _tokenApprovals;
mapping(address => mapping(address => bool)) private
_operatorApprovals;
```

```
event Transfer(address indexed from, address indexed to, uint256
indexed tokenId);
event Approval(address indexed owner, address indexed approved,
uint256 indexed tokenId);
event ApprovalForAll(address indexed owner, address indexed
operator, bool approved);
```

```
constructor(string memory tokenName, string memory
tokenSymbol) {
    name = tokenName;
    symbol = tokenSymbol;
}
```

```
function totalSupply() public view returns (uint256) {
    return currentIndex;
}
```

```
function ownerOf(uint256 tokenId) public view returns (address) {
    return ownershipOf(tokenId).addr;
}
```

```
function ownershipOf(uint256 tokenId) internal view returns
(TokenOwnership memory) {
    require(tokenId < currentIndex);
```

```
for (uint256 curr = tokenId; curr >= 0; curr--) {
    TokenOwnership memory ownership = _ownerships[curr];
    if (ownership.addr != address(0)) {
        return ownership;
    }
}
```

```
revert("Owner query for nonexistent token");
}
```

```
function balanceOf(address owner) public view returns (uint256) {  
    require(owner != address(0));  
    return _balances[owner];  
}
```

```
function _mint(address to, uint256 quantity) internal {  
    require(to != address(0));  
    require(quantity > 0);
```

```
    uint256 startTokenId = currentIndex;
```

```
    _balances[to] += quantity;  
    _ownerships[startTokenId] = TokenOwnership(to,  
    uint64(block.timestamp));
```

```
    for (uint256 i = 0; i < quantity; i++) {  
        emit Transfer(address(0), to, startTokenId + i);  
    }
```

```
    currentIndex += quantity;  
}
```

```
function transferFrom(address from, address to, uint256 tokenId)  
public {  
    TokenOwnership memory prevOwnership =  
    ownershipOf(tokenId);
```

```
    require(prevOwnership.addr == from);  
    require(  
        msg.sender == from ||  
        getApproved(tokenId) == msg.sender ||  
        isApprovedForAll(from, msg.sender)  
    );  
    require(to != address(0));
```

```
    _tokenApprovals[tokenId] = address(0);
```

```
    _balances[from] -= 1;  
    _balances[to] += 1;  
    _ownerships[tokenId] = TokenOwnership(to,
```

```
uint64(block.timestamp));
```

```
uint256 nextTokenId = tokenId + 1;  
if (_ownerships[nextTokenId].addr == address(0) && nextTokenId  
< currentIndex) {  
    _ownerships[nextTokenId] = prevOwnership;  
}
```

```
emit Transfer(from, to, tokenId);  
}
```

```
function approve(address to, uint256 tokenId) public {  
    address owner = ownerOf(tokenId);  
    require(to != owner);  
    require(msg.sender == owner || isApprovedForAll(owner,  
msg.sender));
```

```
    _tokenApprovals[tokenId] = to;  
    emit Approval(owner, to, tokenId);  
}
```

```
function getApproved(uint256 tokenId) public view returns  
(address) {  
    require(tokenId < currentIndex);  
    return _tokenApprovals[tokenId];  
}
```

```
function setApprovalForAll(address operator, bool approved)  
public {  
    require(operator != msg.sender);  
    _operatorApprovals[msg.sender][operator] = approved;  
    emit ApprovalForAll(msg.sender, operator, approved);  
}
```

```
function isApprovedForAll(address owner, address operator) public  
view returns (bool) {  
    return _operatorApprovals[owner][operator];  
}  
}
```

Key insight: Instead of writing ownership for each token in a batch, write once and let later tokens inherit. Reading ownership walks backward until finding a set owner. Trades read gas for write gas -- minting is much cheaper, first transfer slightly more expensive.

Other Optimizations

```
contract GasOptimizedNFT {
    uint256 private constant MAX_SUPPLY = 10000;
    uint256 private constant MINT_PRICE = 0.08 ether;

    uint256 public totalSupply;
    address public immutable owner;

    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;

    constructor() {
        owner = msg.sender;
    }

    function mint(uint256 quantity) external payable {
        require(msg.value >= MINT_PRICE * quantity);

        uint256 supply = totalSupply;
        require(supply + quantity <= MAX_SUPPLY);

        unchecked {
            for (uint256 i; i < quantity; ++i) {
                _owners[supply + i] = msg.sender;
            }
            _balances[msg.sender] += quantity;
            totalSupply = supply + quantity;
        }
    }
}
```

Optimizations applied:

- Constants instead of storage variables (no SLOAD)
- Immutable for one-time set values

- Unchecked arithmetic where safe
- Pre-increment (+ + i) instead of post-increment
- Cache storage reads in memory

SECTION 8: ROYALTY STANDARD (EIP-2981)

EIP-2981 standardizes royalty information:

```
interface IERC2981 {  
    function royaltyInfo(  
        uint256 tokenId,  
        uint256 salePrice  
    ) external view returns (address receiver, uint256 royaltyAmount);  
}
```

```
contract NFTWithRoyalties is ERC721 {  
    address public royaltyReceiver;  
    uint256 public royaltyPercent;
```

```
    uint256 private constant ROYALTY_DENOMINATOR = 10000;
```

```
    address public owner;  
    uint256 public totalSupply;
```

```
    constructor(  
        string memory tokenName,  
        string memory tokenSymbol,  
        address royaltyAddress,  
        uint256 royaltyBps  
    ) ERC721(tokenName, tokenSymbol) {  
        owner = msg.sender;  
        royaltyReceiver = royaltyAddress;  
        royaltyPercent = royaltyBps;  
    }
```

```
    function royaltyInfo(  
        uint256 tokenId,  
        uint256 salePrice  
    ) external view returns (address receiver, uint256 royaltyAmount)
```



```

{
    require(_owners[tokenId] != address(0));
    return (royaltyReceiver, (salePrice * royaltyPercent) /
ROYALTY_DENOMINATOR);
}

function setRoyaltyInfo(address receiver, uint256 percent) external
{
    require(msg.sender == owner);
    require(percent <= 1000);
    royaltyReceiver = receiver;
    royaltyPercent = percent;
}

function supportsInterface(bytes4 interfaceId) public pure override
returns (bool) {
    return
interfaceId == 0x2a55205a ||
super.supportsInterface(interfaceId);
}
}

```

Per-token royalties for collaborations:

```

contract NFTWithPerTokenRoyalties is ERC721 {
    struct RoyaltyInfo {
        address receiver;
        uint96 percent;
    }

    RoyaltyInfo private defaultRoyalty;
    mapping(uint256 => RoyaltyInfo) private tokenRoyalties;

    uint256 private constant ROYALTY_DENOMINATOR = 10000;

    address public owner;
    uint256 public totalSupply;

    constructor(
        string memory tokenName,

```

```

string memory tokenSymbol
) ERC721(tokenName, tokenSymbol) {
owner = msg.sender;
}

```

```

function royaltyInfo(
uint256 tokenId,
uint256 salePrice
) external view returns (address, uint256) {
RoyaltyInfo memory royalty = tokenRoyalties[tokenId];

```

```

if (royalty.receiver == address(0)) {
royalty = defaultRoyalty;
}

```

```

uint256 royaltyAmount = (salePrice * royalty.percent) /
ROYALTY_DENOMINATOR;
return (royalty.receiver, royaltyAmount);
}

```

```

function setDefaultRoyalty(address receiver, uint96 percent)
external {
require(msg.sender == owner);
defaultRoyalty = RoyaltyInfo(receiver, percent);
}

```

```

function setTokenRoyalty(uint256 tokenId, address receiver, uint96
percent) external {
require(msg.sender == owner);
tokenRoyalties[tokenId] = RoyaltyInfo(receiver, percent);
}
}

```

SECTION 9: FRONTEND INTEGRATION

Connecting NFT contracts to user interfaces:

```

import { ethers } from 'ethers'

```

```

const NFT_ABI = [
  'function name() view returns (string)',
  'function symbol() view returns (string)',
  'function totalSupply() view returns (uint256)',
  'function maxSupply() view returns (uint256)',
  'function mintPrice() view returns (uint256)',
  'function mintingEnabled() view returns (bool)',
  'function balanceOf(address) view returns (uint256)',
  'function ownerOf(uint256) view returns (address)',
  'function tokenURI(uint256) view returns (string)',
  'function mint(uint256) payable',
  'function whitelistMint(uint256, uint256, bytes32[]) payable',
  'event Transfer(address indexed from, address indexed to, uint256 indexed tokenId)'
]

```

```

class NFTMinter {
  constructor(contractAddress, provider) {
    this.contract = new ethers.Contract(contractAddress, NFT_ABI,
    provider)
    this.provider = provider
  }

```

```

  async getCollectionInfo() {
    const [name, symbol, totalSupply, maxSupply, mintPrice,
    mintingEnabled] = await Promise.all([
      this.contract.name(),
      this.contract.symbol(),
      this.contract.totalSupply(),
      this.contract.maxSupply(),
      this.contract.mintPrice(),
      this.contract.mintingEnabled()
    ])

```

```

    return {
      name,
      symbol,
      totalSupply: Number(totalSupply),
      maxSupply: Number(maxSupply),
      mintPrice: ethers.formatEther(mintPrice),

```

```
    mintingEnabled,  
    remaining: Number(maxSupply) - Number(totalSupply)  
  }  
}
```

```
async mint(signer, quantity) {  
  const mintPrice = await this.contract.mintPrice()  
  const totalCost = mintPrice * BigInt(quantity)
```

```
  const contractWithSigner = this.contract.connect(signer)  
  const tx = await contractWithSigner.mint(quantity, { value:  
totalCost })
```

```
  return tx.wait()  
}
```

```
async whitelistMint(signer, quantity, maxQuantity, proof) {  
  const mintPrice = await this.contract.mintPrice()  
  const totalCost = mintPrice * BigInt(quantity)
```

```
  const contractWithSigner = this.contract.connect(signer)  
  const tx = await contractWithSigner.whitelistMint(quantity,  
maxQuantity, proof, { value: totalCost })
```

```
  return tx.wait()  
}
```

```
async getOwnedTokens(ownerAddress) {  
  const balance = await this.contract.balanceOf(ownerAddress)  
  const tokens = []
```

```
  const filter = this.contract.filters.Transfer(null, ownerAddress)  
  const events = await this.contract.queryFilter(filter)
```

```
  for (const event of events) {  
    const tokenId = event.args.tokenId  
    const currentOwner = await this.contract.ownerOf(tokenId)
```

```
    if (currentOwner.toLowerCase() ===  
ownerAddress.toLowerCase()) {
```

```
tokens.push(Number(tokenId))
}
}
```

```
return tokens
}
```

```
async getTokenMetadata(tokenId) {
  const uri = await this.contract.tokenURI(tokenId)
  return this.fetchMetadata(uri)
}
```

```
async fetchMetadata(uri) {
  let url = uri
```

```
  if (uri.startsWith('ipfs://')) {
    url = uri.replace('ipfs://', 'https://ipfs.io/ipfs/')
  }
```

```
  if (uri.startsWith('ar://')) {
    url = uri.replace('ar://', 'https://arweave.net/')
  }
```

```
  const response = await fetch(url)
  return response.json()
}
```

```
async estimateMintGas(quantity) {
  const mintPrice = await this.contract.mintPrice()
  const totalCost = mintPrice * BigInt(quantity)
```

```
  try {
    const gas = await this.contract.mint.estimateGas(quantity, { value:
totalCost })
    const feeData = await this.provider.getFeeData()
    const gasCost = gas * feeData.gasPrice
```

```
  return {
    gasUnits: Number(gas),
    gasCostWei: gasCost,
```

```

gasCostEth: ethers.formatEther(gasCost)
}
} catch (error) {
return { error: error.message }
}
}
}
}

```

SECTION 10: TESTING NFT CONTRACTS

Comprehensive testing ensures contract correctness:

```

const { expect } = require('chai')
const { ethers } = require('hardhat')

```

```

describe('MyNFTCollection', function() {
  let nft
  let owner
  let user1
  let user2

```

```

  const NAME = 'Test NFT'
  const SYMBOL = 'TNFT'
  const MAX_SUPPLY = 100
  const MINT_PRICE = ethers.parseEther('0.1')
  const MAX_PER_WALLET = 5

```

```

  beforeEach(async function() {
    [owner, user1, user2] = await ethers.getSigners()

```

```

    const NFT = await ethers.getContractFactory('MyNFTCollection')
    nft = await NFT.deploy(
      NAME,
      SYMBOL,
      MAX_SUPPLY,
      MINT_PRICE,
      MAX_PER_WALLET,
      'ipfs://hidden/'
    )

```

```
})
```

```
describe('Deployment', function() {  
  it('should set correct name and symbol', async function() {  
    expect(await nft.name()).to.equal(NAME)  
    expect(await nft.symbol()).to.equal(SYMBOL)  
  })  
})
```

```
it('should set correct max supply', async function() {  
  expect(await nft.maxSupply()).to.equal(MAX_SUPPLY)  
})
```

```
it('should set owner correctly', async function() {  
  expect(await nft.owner()).to.equal(owner.address)  
})  
})
```

```
describe('Minting', function() {  
  beforeEach(async function() {  
    await nft.enableMinting(true)  
  })  
})
```

```
it('should mint successfully with correct payment', async function()  
{  
  await nft.connect(user1).mint(1, { value: MINT_PRICE })  
  
  expect(await nft.balanceOf(user1.address)).to.equal(1)  
  expect(await nft.ownerOf(1)).to.equal(user1.address)  
})
```

```
it('should mint multiple tokens', async function() {  
  const quantity = 3  
  await nft.connect(user1).mint(quantity, { value: MINT_PRICE *  
    BigInt(quantity) })
```

```
  expect(await nft.balanceOf(user1.address)).to.equal(quantity)  
})
```

```
it('should reject insufficient payment', async function() {  
  await expect(  

```

```
nft.connect(user1).mint(1, { value: MINT_PRICE - 1n })  
.to.be.reverted  
})
```

```
it('should enforce per-wallet limit', async function() {  
  await nft.connect(user1).mint(MAX_PER_WALLET, {  
    value: MINT_PRICE * BigInt(MAX_PER_WALLET)  
  })  
})
```

```
await expect(  
  nft.connect(user1).mint(1, { value: MINT_PRICE })  
)  
.to.be.reverted  
})
```

```
it('should reject when minting is disabled', async function() {  
  await nft.enableMinting(false)
```

```
  await expect(  
    nft.connect(user1).mint(1, { value: MINT_PRICE })  
  ).to.be.reverted  
})  
})
```

```
describe('Transfers', function() {  
  beforeEach(async function() {  
    await nft.enableMinting(true)  
    await nft.connect(user1).mint(1, { value: MINT_PRICE })  
  })
```

```
  it('should transfer from owner', async function() {  
    await nft.connect(user1).transferFrom(user1.address,  
    user2.address, 1)
```

```
  expect(await nft.ownerOf(1)).to.equal(user2.address)  
})
```

```
  it('should reject transfer from non-owner', async function() {  
    await expect(  
      nft.connect(user2).transferFrom(user1.address, user2.address, 1)  
    ).to.be.reverted
```



```
})
```

```
it('should allow transfer after approval', async function() {  
  await nft.connect(user1).approve(user2.address, 1)  
  await nft.connect(user2).transferFrom(user1.address,  
user2.address, 1)
```

```
  expect(await nft.ownerOf(1)).to.equal(user2.address)  
})  
})
```

```
describe('Reveal', function() {  
  beforeEach(async function() {  
    await nft.enableMinting(true)  
    await nft.connect(user1).mint(1, { value: MINT_PRICE })  
  })
```

```
  it('should return hidden URI before reveal', async function() {  
    const uri = await nft.tokenURI(1)  
    expect(uri).to.equal('ipfs://hidden/')  
  })
```

```
  it('should return actual URI after reveal', async function() {  
    await nft.reveal('ipfs://revealed/')  
    const uri = await nft.tokenURI(1)  
    expect(uri).to.equal('ipfs://revealed/1.json')  
  })  
})
```

```
describe('Owner Functions', function() {  
  it('should allow owner to withdraw', async function() {  
    await nft.enableMinting(true)  
    await nft.connect(user1).mint(5, { value: MINT_PRICE * 5n })
```

```
    const balanceBefore = await  
ethers.provider.getBalance(owner.address)  
    await nft.withdraw()  
    const balanceAfter = await  
ethers.provider.getBalance(owner.address)
```

```
expect(balanceAfter).to.be.greaterThan(balanceBefore)
})
```

```
it('should allow owner to mint for free', async function() {
  await nft.ownerMint(user1.address, 10)
```

```
expect(await nft.balanceOf(user1.address)).to.equal(10)
})
```

```
it('should reject non-owner calling owner functions', async
function() {
  await expect(nft.connect(user1).withdraw()).to.be.reverted
  await expect(nft.connect(user1).ownerMint(user1.address,
1)).to.be.reverted
})
})
})
```

Run tests with:

`npx hardhat test`

ERC-721 provides the foundation. Every NFT application builds on these concepts -- ownership, transfers, approvals, metadata. Master this standard and you can build anything the NFT space requires.

CHAPTER 3: ERC-1155 MULTI-TOKEN STANDARD

ERC-721 handles non-fungible tokens. ERC-20 handles fungible tokens. But what if you need both in one contract? What if you have game items where swords are fungible (any sword equals another) but legendary weapons are unique? What if you want to transfer 100 different items in one transaction instead of 100 separate transactions?

ERC-1155 solves these problems. Created by the Enjin team, it's a multi-token standard that handles any combination of fungible and non-fungible tokens in a single contract. Games use it extensively. Any application with multiple asset types benefits from its efficiency.

SECTION 1: WHY ERC-1155 EXISTS

The problems with separate standards:

Gas Inefficiency

Transferring 10 different ERC-721 tokens requires 10 transactions. Each transaction has base gas costs (21,000 gas minimum) plus contract execution. Batch operations are impossible.

ERC-1155 enables batch transfers. Send 100 different token types in one transaction. The gas savings compound dramatically.

Contract Proliferation

A game with 50 item types would need:

- One ERC-20 contract for in-game currency
- One ERC-721 contract for unique items
- Possibly more contracts for different item categories

With ERC-1155, one contract handles everything. Simpler deployment, simpler management, simpler integration.

Atomic Transactions

Swapping items between players with ERC-721 requires careful sequencing or escrow contracts. With ERC-1155, batch transfers enable atomic swaps -- all items move or none do.

The Trade-off

ERC-1155 adds complexity. The standard is harder to understand. Fewer tools support it natively (though major ones do). For simple NFT collections, ERC-721 remains appropriate. Use ERC-1155 when you need its capabilities.

SECTION 2: THE STANDARD INTERFACE

ERC-1155 defines these core functions:

```
interface IERC1155 {  
    function balanceOf(address account, uint256 id) external view  
    returns (uint256);
```

```
    function balanceOfBatch(  
        address[] calldata accounts,  
        uint256[] calldata ids  
    ) external view returns (uint256[] memory);
```

```
    function setApprovalForAll(address operator, bool approved)  
    external;
```

```
    function isApprovedForAll(address account, address operator)  
    external view returns (bool);
```

```
    function safeTransferFrom(  
        address from,  
        address to,  
        uint256 id,  
        uint256 amount,  
        bytes calldata data  
    ) external;
```

```
    function safeBatchTransferFrom(  
        address from,  
        address to,  
        uint256[] calldata ids,  
        uint256[] calldata amounts,  
        bytes calldata data  
    ) external;  
}
```

Required Events

```
event TransferSingle(  
    address indexed operator,  
    address indexed from,  
    address indexed to,
```

```
uint256 id,  
uint256 value  
);
```

```
event TransferBatch(  
    address indexed operator,  
    address indexed from,  
    address indexed to,  
    uint256[] ids,  
    uint256[] values  
);
```

```
event ApprovalForAll(address indexed account, address indexed  
operator, bool approved);
```

```
event URI(string value, uint256 indexed id);
```

Key Differences from ERC-721

No `ownerOf` function: Multiple addresses can own the same token ID (if fungible). Ownership is tracked by balance, not single owner.

Balance includes amount: `balanceOf` returns how many of token ID X an address owns, not just whether they own it.

Batch operations: Core functions support multiple tokens simultaneously.

Single approval type: Only operator approval (like `setApprovalForAll`), no per-token approval.

SECTION 3: IMPLEMENTING ERC-1155

Complete implementation from scratch:

```
contract ERC1155 {  
    mapping(uint256 => mapping(address => uint256)) private  
    _balances;  
    mapping(address => mapping(address => bool)) private
```

```

_operatorApprovals;

string private _uri;

event TransferSingle(
    address indexed operator,
    address indexed from,
    address indexed to,
    uint256 id,
    uint256 value
);

event TransferBatch(
    address indexed operator,
    address indexed from,
    address indexed to,
    uint256[] ids,
    uint256[] values
);

event ApprovalForAll(address indexed account, address indexed
operator, bool approved);
event URI(string value, uint256 indexed id);

constructor(string memory uri) {
    _uri = uri;
}

function uri(uint256) public view returns (string memory) {
    return _uri;
}

function balanceOf(address account, uint256 id) public view
returns (uint256) {
    require(account != address(0));
    return _balances[id][account];
}

function balanceOfBatch(
    address[] memory accounts,

```

```
uint256[] memory ids
) public view returns (uint256[] memory) {
    require(accounts.length == ids.length);
```

```
uint256[] memory batchBalances = new uint256[]
(accounts.length);
```

```
for (uint256 i = 0; i < accounts.length; i++) {
    batchBalances[i] = balanceOf(accounts[i], ids[i]);
}
```

```
return batchBalances;
}
```

```
function setApprovalForAll(address operator, bool approved)
public {
    require(operator != msg.sender);
    _operatorApprovals[msg.sender][operator] = approved;
    emit ApprovalForAll(msg.sender, operator, approved);
}
```

```
function isApprovedForAll(address account, address operator)
public view returns (bool) {
    return _operatorApprovals[account][operator];
}
```

```
function safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) public {
    require(from == msg.sender || isApprovedForAll(from,
msg.sender));
    require(to != address(0));
```

```
_balances[id][from] -= amount;
_balances[id][to] += amount;
```

```
emit TransferSingle(msg.sender, from, to, id, amount);
```

```
_doSafeTransferAcceptanceCheck(msg.sender, from, to, id, amount,  
data);  
}
```

```
function safeBatchTransferFrom(  
address from,  
address to,  
uint256[] memory ids,  
uint256[] memory amounts,  
bytes memory data  
) public {  
require(ids.length == amounts.length);  
require(from == msg.sender || isApprovedForAll(from,  
msg.sender));  
require(to != address(0));
```

```
for (uint256 i = 0; i < ids.length; i++) {  
_balances[ids[i]][from] -= amounts[i];  
_balances[ids[i]][to] += amounts[i];  
}
```

```
emit TransferBatch(msg.sender, from, to, ids, amounts);
```

```
_doSafeBatchTransferAcceptanceCheck(msg.sender, from, to, ids,  
amounts, data);  
}
```

```
function _mint(address to, uint256 id, uint256 amount, bytes  
memory data) internal {  
require(to != address(0));
```

```
_balances[id][to] += amount;
```

```
emit TransferSingle(msg.sender, address(0), to, id, amount);
```

```
_doSafeTransferAcceptanceCheck(msg.sender, address(0), to, id,  
amount, data);  
}
```



```

function _mintBatch(
  address to,
  uint256[] memory ids,
  uint256[] memory amounts,
  bytes memory data
) internal {
  require(to != address(0));
  require(ids.length == amounts.length);

  for (uint256 i = 0; i < ids.length; i++) {
    _balances[ids[i]][to] += amounts[i];
  }

  emit TransferBatch(msg.sender, address(0), to, ids, amounts);

  _doSafeBatchTransferAcceptanceCheck(msg.sender, address(0), to,
  ids, amounts, data);
}

function _burn(address from, uint256 id, uint256 amount) internal
{
  require(from != address(0));

  _balances[id][from] -= amount;

  emit TransferSingle(msg.sender, from, address(0), id, amount);
}

function _burnBatch(
  address from,
  uint256[] memory ids,
  uint256[] memory amounts
) internal {
  require(from != address(0));
  require(ids.length == amounts.length);

  for (uint256 i = 0; i < ids.length; i++) {
    _balances[ids[i]][from] -= amounts[i];
  }
}

```

```
emit TransferBatch(msg.sender, from, address(0), ids, amounts);  
}
```

```
function _doSafeTransferAcceptanceCheck(  
    address operator,  
    address from,  
    address to,  
    uint256 id,  
    uint256 amount,  
    bytes memory data  
) private {  
    if (to.code.length > 0) {  
        try IERC1155Receiver(to).onERC1155Received(operator, from, id,  
            amount, data) returns (bytes4 response) {  
            if (response != IERC1155Receiver.onERC1155Received.selector) {  
                revert("ERC1155: rejected tokens");  
            }  
        } catch Error(string memory reason) {  
            revert(reason);  
        } catch {  
            revert("ERC1155: transfer to non-receiver");  
        }  
    }  
}
```

```
function _doSafeBatchTransferAcceptanceCheck(  
    address operator,  
    address from,  
    address to,  
    uint256[] memory ids,  
    uint256[] memory amounts,  
    bytes memory data  
) private {  
    if (to.code.length > 0) {  
        try IERC1155Receiver(to).onERC1155BatchReceived(operator,  
            from, ids, amounts, data) returns (bytes4 response) {  
            if (response !=  
                IERC1155Receiver.onERC1155BatchReceived.selector) {  
                revert("ERC1155: rejected tokens");  
            }  
        }  
    }  
}
```

```

}
} catch Error(string memory reason) {
revert(reason);
} catch {
revert("ERC1155: transfer to non-receiver");
}
}
}

```

```

function supportsInterface(bytes4 interfaceId) public pure returns
(bool) {
return
interfaceId == 0xd9b67a26 ||
interfaceId == 0x0e89341c ||
interfaceId == 0x01ffc9a7;
}
}

```

```

interface IERC1155Receiver {
function onERC1155Received(
address operator,
address from,
uint256 id,
uint256 value,
bytes calldata data
) external returns (bytes4);
}

```

```

function onERC1155BatchReceived(
address operator,
address from,
uint256[] calldata ids,
uint256[] calldata values,
bytes calldata data
) external returns (bytes4);
}

```

SECTION 4: FUNGIBLE VS NON-FUNGIBLE IN ERC-1155

The same contract handles both types:

Fungible tokens: Multiple units exist. Token ID 1 might have 1,000,000 units. Any unit is interchangeable.

Non-fungible tokens: Exactly one unit exists. Token ID 1000 has supply of 1. Unique.

Semi-fungible tokens: Limited supply greater than 1. Token ID 500 might have 100 units -- fungible among themselves but distinct from other IDs.

```
contract GameItems is ERC1155 {
    uint256 public constant GOLD = 0;
    uint256 public constant SILVER = 1;
    uint256 public constant IRON_SWORD = 2;
    uint256 public constant IRON_SHIELD = 3;

    uint256 private _currentTokenId = 1000;

    mapping(uint256 => uint256) public tokenSupply;
    mapping(uint256 => uint256) public maxSupply;
    mapping(uint256 => bool) public isNonFungible;

    address public owner;

    constructor() ERC1155("https://game.example/api/item/{id}.json") {
        owner = msg.sender;

        maxSupply[GOLD] = type(uint256).max;
        maxSupply[SILVER] = type(uint256).max;
        maxSupply[IRON_SWORD] = 10000;
        maxSupply[IRON_SHIELD] = 10000;
    }

    function mintFungible(address to, uint256 id, uint256 amount)
    external {
        require(msg.sender == owner);
        require(!isNonFungible[id]);
        require(tokenSupply[id] + amount <= maxSupply[id]);
```

```

tokenSupply[id] += amount;
_mint(to, id, amount, "");
}

```

```

function mintNonFungible(address to) external returns (uint256) {
    require(msg.sender == owner);

```

```

    uint256 newTokenId = _currentTokenId;
    _currentTokenId++;

```

```

    isNonFungible[newTokenId] = true;
    maxSupply[newTokenId] = 1;
    tokenSupply[newTokenId] = 1;

```

```

    _mint(to, newTokenId, 1, "");

```

```

    return newTokenId;
}

```

```

function mintLimitedEdition(address to, uint256 id, uint256
amount, uint256 maxEdition) external {
    require(msg.sender == owner);
    require(tokenSupply[id] == 0);
    require(amount <= maxEdition);

```

```

    maxSupply[id] = maxEdition;
    tokenSupply[id] = amount;

```

```

    _mint(to, id, amount, "");
}

```

```

function isFungible(uint256 id) public view returns (bool) {
    return maxSupply[id] > 1 && !isNonFungible[id];
}
}

```

SECTION 5: GAME ASSET PATTERNS

Games are the primary ERC-1155 use case. Here's a complete game item system:

```
contract GameItemSystem is ERC1155 {
    struct ItemType {
        string name;
        uint8 rarity;
        uint256 maxSupply;
        uint256 currentSupply;
        bool tradeable;
        bool consumable;
    }

    mapping(uint256 => ItemType) public itemTypes;
    mapping(uint256 => mapping(string => uint256)) public
    itemAttributes;

    uint256 public nextItemTypeId;

    address public owner;
    address public gameServer;

    mapping(address => bool) public approvedMinters;

    event ItemTypeCreated(uint256 indexed typeId, string name, uint8
    rarity);
    event ItemConsumed(address indexed user, uint256 indexed
    typeId, uint256 amount);

    constructor() ERC1155("https://game.example/items/{id}.json") {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    modifier onlyMinter() {
        require(approvedMinters[msg.sender] || msg.sender == owner);
```

```
_;  
}
```

```
function createItemType(  
string memory name,  
uint8 rarity,  
uint256 maxSupply,  
bool tradeable,  
bool consumable  
) external onlyOwner returns (uint256) {  
uint256 typeId = nextItemId;  
nextItemId ++;  

```

```
itemTypes[typeId] = ItemType({  
name: name,  
rarity: rarity,  
maxSupply: maxSupply,  
currentSupply: 0,  
tradeable: tradeable,  
consumable: consumable  
});
```

```
emit ItemCreated(typeId, name, rarity);  
return typeId;  
}
```

```
function setItemAttribute(uint256 typeId, string memory attribute,  
uint256 value) external onlyOwner {  
itemAttributes[typeId][attribute] = value;  
}
```

```
function mint(address to, uint256 typeId, uint256 amount) external  
onlyMinter {  
ItemType storage item = itemTypes[typeId];  
require(bytes(item.name).length > 0);  
require(item.currentSupply + amount <= item.maxSupply);  
  
item.currentSupply += amount;  
_mint(to, typeId, amount, "");  
}
```

```

function mintBatch(
  address to,
  uint256[] memory typeIds,
  uint256[] memory amounts
) external onlyMinter {
  require(typeIds.length == amounts.length);

  for (uint256 i = 0; i < typeIds.length; i++) {
    ItemType storage item = itemTypes[typeIds[i]];
    require(bytes(item.name).length > 0);
    require(item.currentSupply + amounts[i] <= item.maxSupply);
    item.currentSupply += amounts[i];
  }

  _mintBatch(to, typeIds, amounts, "");
}

```

```

function consume(uint256 typeId, uint256 amount) external {
  ItemType storage item = itemTypes[typeId];
  require(item.consumable);
  require(balanceOf(msg.sender, typeId) >= amount);

  _burn(msg.sender, typeId, amount);
  emit ItemConsumed(msg.sender, typeId, amount);
}

```

```

function safeTransferFrom(
  address from,
  address to,
  uint256 id,
  uint256 amount,
  bytes memory data
) public override {
  require(itemTypes[id].tradeable);
  super.safeTransferFrom(from, to, id, amount, data);
}

```

```

function safeBatchTransferFrom(
  address from,

```



```

address to,
uint256[] memory ids,
uint256[] memory amounts,
bytes memory data
) public override {
for (uint256 i = 0; i < ids.length; i++) {
require(itemTypes[ids[i]].tradeable);
}
super.safeBatchTransferFrom(from, to, ids, amounts, data);
}

```

```

function setApprovedMinter(address minter, bool approved)
external onlyOwner {
approvedMinters[minter] = approved;
}

```

```

function getItemType(uint256 typeId) external view returns
(ItemType memory) {
return itemTypes[typeId];
}

```

```

function getAttribute(uint256 typeId, string memory attribute)
external view returns (uint256) {
return itemAttributes[typeId][attribute];
}
}

```

Loot Box Implementation

```

contract LootBoxSystem is ERC1155 {
uint256 public constant COMMON_BOX = 0;
uint256 public constant RARE_BOX = 1;
uint256 public constant LEGENDARY_BOX = 2;

```

```

struct LootTable {
uint256[] itemIds;
uint256[] weights;
uint256 totalWeight;
}

```

```
mapping(uint256 => LootTable) private lootTables;
mapping(uint256 => uint256) public boxPrices;
```

```
address public owner;
uint256 private nonce;
```

```
event BoxOpened(address indexed user, uint256 boxType,
uint256[] itemsReceived, uint256[] amounts);
```

```
constructor() ERC1155("https://game.example/loot/{id}.json") {
    owner = msg.sender;
}
```

```
function setLootTable(
uint256 boxType,
uint256[] memory itemIds,
uint256[] memory weights
) external {
    require(msg.sender == owner);
    require(itemIds.length == weights.length);
```

```
    uint256 totalWeight = 0;
    for (uint256 i = 0; i < weights.length; i++) {
        totalWeight += weights[i];
    }
```

```
    lootTables[boxType] = LootTable({
        itemIds: itemIds,
        weights: weights,
        totalWeight: totalWeight
    });
}
```

```
function buyBox(uint256 boxType, uint256 amount) external
payable {
    require(msg.value >= boxPrices[boxType] * amount);
    _mint(msg.sender, boxType, amount, "");
}
```

```
function openBox(uint256 boxType, uint256 amount) external {
```

```
require(balanceOf(msg.sender, boxType) >= amount);
```

```
_burn(msg.sender, boxType, amount);
```

```
uint256[] memory receivedItems = new uint256[](amount);
```

```
uint256[] memory receivedAmounts = new uint256[](amount);
```

```
for (uint256 i = 0; i < amount; i++) {
```

```
    uint256 itemId = _rollLoot(boxType);
```

```
    receivedItems[i] = itemId;
```

```
    receivedAmounts[i] = 1;
```

```
}
```

```
_mintBatch(msg.sender, receivedItems, receivedAmounts, "");
```

```
emit BoxOpened(msg.sender, boxType, receivedItems,  
receivedAmounts);
```

```
}
```

```
function _rollLoot(uint256 boxType) private returns (uint256) {
```

```
    LootTable storage table = lootTables[boxType];
```

```
    require(table.totalWeight > 0);
```

```
    uint256 roll = _random() % table.totalWeight;
```

```
    uint256 cumulative = 0;
```

```
    for (uint256 i = 0; i < table.itemIds.length; i++) {
```

```
        cumulative += table.weights[i];
```

```
        if (roll < cumulative) {
```

```
            return table.itemIds[i];
```

```
        }
```

```
    }
```

```
    return table.itemIds[table.itemIds.length - 1];
```

```
}
```

```
function _random() private returns (uint256) {
```

```
    nonce++;
```

```
    return uint256(keccak256(abi.encodePacked(block.timestamp,  
msg.sender, nonce)));
```

```
}
```

```
function setBoxPrice(uint256 boxType, uint256 price) external {  
    require(msg.sender == owner);  
    boxPrices[boxType] = price;  
}  
}
```

Note: The randomness in `_random()` is predictable and exploitable. Production games should use Chainlink VRF or similar secure randomness.

SECTION 6: BATCH OPERATIONS

Batch operations are ERC-1155's efficiency advantage:

```
contract BatchOperations is ERC1155 {  
    address public owner;  
  
    constructor() ERC1155("https://example.com/{id}.json") {  
        owner = msg.sender;  
    }  
  
    function airdropSingle(  
        address[] memory recipients,  
        uint256 tokenId,  
        uint256 amount  
    ) external {  
        require(msg.sender == owner);  
  
        for (uint256 i = 0; i < recipients.length; i++) {  
            _mint(recipients[i], tokenId, amount, "");  
        }  
    }  
  
    function airdropMultiple(  
        address[] memory recipients,  
        uint256[] memory tokenIds,  
        uint256[] memory amounts
```

```

) external {
require(msg.sender == owner);
require(recipients.length == tokenIds.length);
require(tokenIds.length == amounts.length);

for (uint256 i = 0; i < recipients.length; i++) {
    _mint(recipients[i], tokenIds[i], amounts[i], "");
}
}

```

```

function airdropBatch(
address[] memory recipients,
uint256[] memory tokenIds,
uint256[] memory amounts
) external {
require(msg.sender == owner);

for (uint256 i = 0; i < recipients.length; i++) {
    _mintBatch(recipients[i], tokenIds, amounts, "");
}
}

```

```

function burnBatch(uint256[] memory tokenIds, uint256[]
memory amounts) external {
    _burnBatch(msg.sender, tokenIds, amounts);
}
}

```

Gas Comparison

Single transfers:

- 10 ERC-721 transfers: ~500,000 gas (50,000 each)
- 10 ERC-1155 single transfers: ~400,000 gas (40,000 each)
- 1 ERC-1155 batch transfer of 10: ~100,000 gas

Batch operations save 75%+ gas for multi-token transfers.

SECTION 7: METADATA FOR ERC-1155

ERC-1155 metadata differs slightly from ERC-721:

```
contract ERC1155WithMetadata is ERC1155 {  
    mapping(uint256 => string) private _tokenURIs;  
    string private _baseURI;
```

```
    address public owner;
```

```
    constructor(string memory baseURI) ERC1155(baseURI) {  
        owner = msg.sender;  
        _baseURI = baseURI;  
    }
```

```
    function uri(uint256 tokenId) public view override returns (string  
memory) {  
        string memory tokenURI = _tokenURIs[tokenId];
```

```
        if (bytes(tokenURI).length > 0) {  
            return tokenURI;  
        }
```

```
        return string(abi.encodePacked(_baseURI, toString(tokenId),  
".json"));  
    }
```

```
    function setTokenURI(uint256 tokenId, string memory tokenURI)  
external {  
    require(msg.sender == owner);  
    _tokenURIs[tokenId] = tokenURI;  
    emit URI(tokenURI, tokenId);  
}
```

```
    function setBaseURI(string memory newBaseURI) external {  
        require(msg.sender == owner);  
        _baseURI = newBaseURI;  
    }
```

```
    function toString(uint256 value) internal pure returns (string  
memory) {  
        if (value == 0) return "0";
```

```

uint256 temp = value;
uint256 digits;
while (temp != 0) {
    digits ++;
    temp /= 10;
}
bytes memory buffer = new bytes(digits);
while (value != 0) {
    digits -= 1;
    buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
    value /= 10;
}
return string(buffer);
}
}

```

ERC-1155 Metadata JSON

```

{
  "name": "Iron Sword",
  "description": "A basic iron sword for beginners",
  "image": "ipfs://QmXxx.../sword.png",
  "decimals": 0,
  "properties": {
    "damage": 10,
    "durability": 100,
    "rarity": "common",
    "type": "weapon"
  }
}

```

The decimals field indicates fungibility:

- decimals: 0 -- non-fungible or semi-fungible (whole units only)
- decimals: 18 -- fungible like ERC-20 tokens

SECTION 8: FRONTEND INTEGRATION

Working with ERC-1155 from JavaScript:

```
import { ethers } from 'ethers'
```

```
const ERC1155_ABI = [  
  'function balanceOf(address account, uint256 id) view returns  
(uint256)',  
  'function balanceOfBatch(address[] accounts, uint256[] ids) view  
returns (uint256[])',  
  'function isApprovedForAll(address account, address operator) view  
returns (bool)',  
  'function setApprovalForAll(address operator, bool approved)',  
  'function safeTransferFrom(address from, address to, uint256 id,  
uint256 amount, bytes data)',  
  'function safeBatchTransferFrom(address from, address to,  
uint256[] ids, uint256[] amounts, bytes data)',  
  'function uri(uint256 id) view returns (string)',  
  'event TransferSingle(address indexed operator, address indexed  
from, address indexed to, uint256 id, uint256 value)',  
  'event TransferBatch(address indexed operator, address indexed  
from, address indexed to, uint256[] ids, uint256[] values)'  
]
```

```
class ERC1155Client {  
  constructor(contractAddress, provider) {  
    this.contract = new ethers.Contract(contractAddress,  
ERC1155_ABI, provider)  
    this.provider = provider  
  }  

```

```
  async getBalance(account, tokenId) {  
    return this.contract.balanceOf(account, tokenId)  
  }
```

```
  async getBalances(account, tokenIds) {  
    const accounts = tokenIds.map(() => account)  
    return this.contract.balanceOfBatch(accounts, tokenIds)  
  }
```

```
  async getAllBalances(account, maxTokenId) {  
    const tokenIds = []  
    const accounts = []
```



```
for (let i = 0; i <= maxTokenId; i++) {  
  tokenIds.push(i)  
  accounts.push(account)  
}
```

```
const balances = await this.contract.balanceOfBatch(accounts,  
tokenIds)
```

```
const result = []  
for (let i = 0; i < balances.length; i++) {  
  if (balances[i] > 0n) {  
    result.push({  
      tokenId: i,  
      balance: balances[i]  
    })  
  }  
}
```

```
return result  
}
```

```
async transfer(signer, to, tokenId, amount) {  
  const contractWithSigner = this.contract.connect(signer)  
  const from = await signer.getAddress()
```

```
  const tx = await contractWithSigner.safeTransferFrom(  
    from,  
    to,  
    tokenId,  
    amount,  
    '0x'  
  )
```

```
  return tx.wait()  
}
```

```
async batchTransfer(signer, to, tokenIds, amounts) {  
  const contractWithSigner = this.contract.connect(signer)  
  const from = await signer.getAddress()
```

```
const tx = await contractWithSigner.safeBatchTransferFrom(
  from,
  to,
  tokenIds,
  amounts,
  '0x'
)
```

```
return tx.wait()
}
```

```
async setApproval(signer, operator, approved) {
  const contractWithSigner = this.contract.connect(signer)
  const tx = await contractWithSigner.setApprovalForAll(operator,
  approved)
  return tx.wait()
}
```

```
async isApproved(owner, operator) {
  return this.contract.isApprovedForAll(owner, operator)
}
```

```
async getMetadata(tokenId) {
  const uri = await this.contract.uri(tokenId)
  return this.fetchMetadata(uri, tokenId)
}
```

```
async fetchMetadata(uri, tokenId) {
  let url = uri.replace('{id}', tokenId.toString().padStart(64, '0'))
```

```
if (url.startsWith('ipfs://')) {
  url = url.replace('ipfs://', 'https://ipfs.io/ipfs/')
}
```

```
const response = await fetch(url)
return response.json()
}
```

```
async getTransferHistory(fromBlock = 0) {
```

```
const singleFilter = this.contract.filters.TransferSingle()
const batchFilter = this.contract.filters.TransferBatch()
```

```
const [singleEvents, batchEvents] = await Promise.all([
  this.contract.queryFilter(singleFilter, fromBlock),
  this.contract.queryFilter(batchFilter, fromBlock)
])
```

```
const transfers = []
```

```
for (const event of singleEvents) {
  transfers.push({
    type: 'single',
    operator: event.args.operator,
    from: event.args.from,
    to: event.args.to,
    tokenId: Number(event.args.id),
    amount: event.args.value,
    blockNumber: event.blockNumber,
    txHash: event.transactionHash
  })
}
```

```
for (const event of batchEvents) {
  transfers.push({
    type: 'batch',
    operator: event.args.operator,
    from: event.args.from,
    to: event.args.to,
    tokenIds: event.args.ids.map(Number),
    amounts: event.args.values,
    blockNumber: event.blockNumber,
    txHash: event.transactionHash
  })
}
```

```
return transfers.sort((a, b) => a.blockNumber - b.blockNumber)
}
}
```

SECTION 9: COMBINING WITH ERC-721

Some projects need both standards. Here's a hybrid approach:

```
contract HybridNFT {
    mapping(uint256 => mapping(address => uint256)) private
    _balances1155;
    mapping(uint256 => address) private _owners721;
    mapping(address => uint256) private _balances721;

    mapping(uint256 => bool) public is721;

    mapping(address => mapping(address => bool)) private
    _operatorApprovals;
    mapping(uint256 => address) private _tokenApprovals;

    address public owner;
    uint256 private _currentTokenId;

    event TransferSingle(address indexed operator, address indexed
    from, address indexed to, uint256 id, uint256 value);
    event Transfer(address indexed from, address indexed to, uint256
    indexed tokenId);
    event ApprovalForAll(address indexed account, address indexed
    operator, bool approved);
    event Approval(address indexed owner, address indexed approved,
    uint256 indexed tokenId);

    constructor() {
        owner = msg.sender;
    }

    function mintFungible(address to, uint256 id, uint256 amount)
    external {
        require(msg.sender == owner);
        require(!is721[id]);

        _balances1155[id][to] += amount;
        emit TransferSingle(msg.sender, address(0), to, id, amount);
    }
}
```

```
}
```

```
function mintNFT(address to) external returns (uint256) {  
    require(msg.sender == owner);
```

```
    uint256 tokenId = _currentTokenId;  
    _currentTokenId ++;
```

```
    is721[tokenId] = true;  
    _owners721[tokenId] = to;  
    _balances721[to] ++;
```

```
    emit Transfer(address(0), to, tokenId);  
    return tokenId;  
}
```

```
function balanceOf(address account, uint256 id) public view  
returns (uint256) {  
    if (is721[id]) {  
        return _owners721[id] == account ? 1 : 0;  
    }  
    return _balances1155[id][account];  
}
```

```
function ownerOf(uint256 tokenId) public view returns (address) {  
    require(is721[tokenId]);  
    address tokenOwner = _owners721[tokenId];  
    require(tokenOwner != address(0));  
    return tokenOwner;  
}
```

```
function safeTransferFrom(  
    address from,  
    address to,  
    uint256 id,  
    uint256 amount,  
    bytes memory data  
) public {  
    require(from == msg.sender || _operatorApprovals[from]  
[msg.sender]);
```

```

if (is721[id]) {
    require(amount == 1);
    require(_owners721[id] == from);
    require(
        msg.sender == from ||
        _operatorApprovals[from][msg.sender] ||
        _tokenApprovals[id] == msg.sender
    );

    _tokenApprovals[id] = address(0);
    _balances721[from]--;
    _balances721[to]++;
    _owners721[id] = to;

    emit Transfer(from, to, id);
} else {
    _balances1155[id][from] -= amount;
    _balances1155[id][to] += amount;
}

emit TransferSingle(msg.sender, from, to, id, amount);
}

function setApprovalForAll(address operator, bool approved)
public {
    _operatorApprovals[msg.sender][operator] = approved;
    emit ApprovalForAll(msg.sender, operator, approved);
}

function approve(address to, uint256 tokenId) public {
    require(is721[tokenId]);
    address tokenOwner = _owners721[tokenId];
    require(msg.sender == tokenOwner ||
        _operatorApprovals[tokenOwner][msg.sender]);

    _tokenApprovals[tokenId] = to;
    emit Approval(tokenOwner, to, tokenId);
}

```

```

function isApprovedForAll(address account, address operator)
public view returns (bool) {
    return _operatorApprovals[account][operator];
}
}

```

SECTION 10: TESTING ERC-1155

Comprehensive tests for ERC-1155 contracts:

```

const { expect } = require('chai')
const { ethers } = require('hardhat')

describe('GameItems', function() {
    let gameItems
    let owner
    let player1
    let player2

    beforeEach(async function() {
        [owner, player1, player2] = await ethers.getSigners()

        const GameItems = await ethers.getContractFactory('GameItems')
        gameItems = await GameItems.deploy()
    })

    describe('Fungible Items', function() {
        it('should mint fungible tokens', async function() {
            await gameItems.mintFungible(player1.address, 0, 1000)

            expect(await gameItems.balanceOf(player1.address,
            0)).to.equal(1000)
        })

        it('should transfer fungible tokens', async function() {
            await gameItems.mintFungible(player1.address, 0, 1000)

            await gameItems.connect(player1).safeTransferFrom(
            player1.address,

```

```
player2.address,  
0,  
500,  
'0x'  
)
```

```
expect(await gameItems.balanceOf(player1.address,  
0)).to.equal(500)  
expect(await gameItems.balanceOf(player2.address,  
0)).to.equal(500)  
})  
})
```

```
describe('Non-Fungible Items', function() {  
  it('should mint unique tokens', async function() {  
    const tx = await gameItems.mintNonFungible(player1.address)  
    const receipt = await tx.wait()
```

```
    expect(await gameItems.balanceOf(player1.address,  
1000)).to.equal(1)  
  })
```

```
  it('should enforce uniqueness', async function() {  
    await gameItems.mintNonFungible(player1.address)
```

```
    expect(await gameItems.isNonFungible(1000)).to.equal(true)  
    expect(await gameItems.maxSupply(1000)).to.equal(1)  
  })  
})
```

```
describe('Batch Operations', function() {  
  it('should query batch balances', async function() {  
    await gameItems.mintFungible(player1.address, 0, 100)  
    await gameItems.mintFungible(player1.address, 1, 200)  
    await gameItems.mintFungible(player1.address, 2, 50)
```

```
    const balances = await gameItems.balanceOfBatch(  
[player1.address, player1.address, player1.address],  
[0, 1, 2]  
)
```



```
expect(balances[0]).to.equal(100)
expect(balances[1]).to.equal(200)
expect(balances[2]).to.equal(50)
})
```

```
it('should batch transfer', async function() {
  await gameItems.mintFungible(player1.address, 0, 100)
  await gameItems.mintFungible(player1.address, 1, 100)
```

```
  await gameItems.connect(player1).safeBatchTransferFrom(
    player1.address,
    player2.address,
    [0, 1],
    [50, 25],
    '0x'
  )
```

```
  expect(await gameItems.balanceOf(player2.address,
0)).to.equal(50)
  expect(await gameItems.balanceOf(player2.address,
1)).to.equal(25)
})
})
```

```
describe('Approvals', function() {
  it('should allow approved operators', async function() {
    await gameItems.mintFungible(player1.address, 0, 100)
    await
gameItems.connect(player1).setApprovalForAll(player2.address,
true)
```

```
    await gameItems.connect(player2).safeTransferFrom(
      player1.address,
      player2.address,
      0,
      50,
      '0x'
    )
```

```
expect(await gameItems.balanceOf(player2.address,  
0)).to.equal(50)  
})
```

```
it('should reject unapproved transfers', async function() {  
  await gameItems.mintFungible(player1.address, 0, 100)
```

```
  
  await expect(  
    gameItems.connect(player2).safeTransferFrom(  
      player1.address,  
      player2.address,  
      0,  
      50,  
      '0x'  
    )  
  ).to.be.reverted  
})  
})
```

```
describe('Supply Limits', function() {  
  it('should enforce max supply', async function() {  
    await expect(  
      gameItems.mintFungible(player1.address, 2, 20000)  
    ).to.be.reverted  
  })  
})
```

```
it('should track current supply', async function() {  
  await gameItems.mintFungible(player1.address, 2, 5000)
```

```
  
  expect(await gameItems.tokenSupply(2)).to.equal(5000)  
})  
})  
})
```

ERC-1155 enables sophisticated token systems impossible with single-type standards. Games, marketplaces, and any application managing multiple asset types benefit from its efficiency and flexibility. The learning curve is worth the capabilities it provides.

CHAPTER 4: METADATA AND IPFS

The token on-chain is just a number. What makes NFT #4523 a blue ape with laser eyes? Metadata. The JSON file that describes what the token represents, what it looks like, and what attributes it has. Without metadata, NFTs are meaningless integers.

This chapter explores how metadata works, how to structure it correctly, and how to store it reliably. IPFS dominates NFT storage because it provides content-addressed permanence without centralized control. You'll understand how IPFS works, how to pin content, and how to ensure your NFT assets survive.

SECTION 1: METADATA STRUCTURE

NFT metadata follows conventions established by OpenSea and adopted industry-wide. Marketplaces parse this JSON to display NFTs correctly.

Standard Fields

```
{
  "name": "Cool Cat #1234",
  "description": "A randomly generated cool cat living on the
Ethereum blockchain.",
  "image": "ipfs://QmXnYz.../1234.png",
  "external_url": "https://coolcats.com/cat/1234",
  "animation_url": "ipfs://QmAbCd.../1234.mp4",
  "background_color": "FFFFFF",
  "attributes": [
    {
      "trait_type": "Background",
      "value": "Blue"
    },
    {
      "trait_type": "Fur",
      "value": "Orange Tabby"
    },
    {
      "trait_type": "Eyes",
      "value": "Laser"
    }
  ]
}
```

```
},
{
  "trait_type": "Hat",
  "value": "Crown"
},
{
  "trait_type": "Power Level",
  "display_type": "number",
  "value": 95
},
{
  "trait_type": "Generation",
  "display_type": "number",
  "value": 1
},
{
  "trait_type": "Birthday",
  "display_type": "date",
  "value": 1609459200
}
]
}
```

Field Explanations

name: The display name for this specific token. Usually includes the collection name and token number.

description: Human-readable description. Can include markdown in some marketplaces. Describes the NFT and collection context.

image: URL to the visual representation. Most critical field. Supports IPFS, Arweave, HTTP, and data URIs.

external_url: Link to view more about this NFT on the project's website. Opens when users click "View Original" on marketplaces.

animation_url: For animated or interactive NFTs. Supports video files (MP4, WebM), audio (MP3, WAV), 3D models (GLB, GLTF), and HTML pages.

background_color: Six-character hex color without the #. Used as background when displaying the NFT.

attributes: Array of trait objects defining properties. Powers rarity calculations and filtering.

Attribute Display Types

Standard traits show as text:

```
{  
  "trait_type": "Fur Color",  
  "value": "Golden"  
}
```

Numeric traits for stats and levels:

```
{  
  "trait_type": "Strength",  
  "display_type": "number",  
  "value": 85  
}
```

Boost percentages show as +X%:

```
{  
  "trait_type": "Speed Boost",  
  "display_type": "boost_percentage",  
  "value": 15  
}
```

Boost numbers show as +X:

```
{  
  "trait_type": "Attack Bonus",  
  "display_type": "boost_number",  
  "value": 25  
}
```

Date traits display as dates:

```
{
  "trait_type": "Created",
  "display_type": "date",
  "value": 1640000000
}
```

Ranking traits for leaderboards:

```
{
  "trait_type": "Rank",
  "display_type": "number",
  "max_value": 10000,
  "value": 42
}
```

SECTION 2: GENERATING METADATA

For collections, metadata generation is automated:

```
import json
import os
from pathlib import Path

class MetadataGenerator:
    def __init__(self, collection_name, description_template,
base_image_uri):
        self.collection_name = collection_name
        self.description_template = description_template
        self.base_image_uri = base_image_uri

    def generate_single(self, token_id, attributes):
        metadata = {
            "name": f"{self.collection_name} #{token_id}",
            "description": self.description_template.format(token_id=token_id),
            "image": f"{self.base_image_uri}/{token_id}.png",
            "attributes": self.format_attributes(attributes)
        }
```

```
return metadata
```

```
def format_attributes(self, attributes):  
    formatted = []  
    for trait_type, value in attributes.items():  
        if isinstance(value, dict):  
            attr = {"trait_type": trait_type, **value}  
        else:  
            attr = {"trait_type": trait_type, "value": value}  
        formatted.append(attr)  
    return formatted
```

```
def generate_collection(self, token_data):  
    metadata_list = []  
    for token_id, attributes in token_data.items():  
        metadata = self.generate_single(token_id, attributes)  
        metadata_list.append((token_id, metadata))  
    return metadata_list
```

```
def save_to_files(self, metadata_list, output_dir):  
    Path(output_dir).mkdir(parents=True, exist_ok=True)
```

```
    for token_id, metadata in metadata_list:  
        filepath = os.path.join(output_dir, f'{token_id}.json')  
        with open(filepath, 'w') as f:  
            json.dump(metadata, f, indent=2)
```

```
    return len(metadata_list)
```

```
def generate_random_collection(size, trait_options):  
    import random
```

```
    token_data = {}
```

```
    for token_id in range(1, size + 1):  
        attributes = {}  
        for trait_type, options in trait_options.items():  
            if isinstance(options, list):  
                attributes[trait_type] = random.choice(options)
```

```

elif isinstance(options, dict):
    if options.get('type') == 'number':
        attributes[trait_type] = {
            "display_type": "number",
            "value": random.randint(options['min'], options['max'])
        }
    token_data[token_id] = attributes

return token_data

```

```

trait_options = {
    "Background": ["Blue", "Red", "Green", "Purple", "Yellow", "Black"],
    "Body": ["Human", "Robot", "Alien", "Zombie", "Ape"],
    "Eyes": ["Normal", "Laser", "Cyclops", "Closed", "Heart"],
    "Mouth": ["Smile", "Frown", "Open", "Tongue", "Fangs"],
    "Hat": ["None", "Cap", "Crown", "Helmet", "Halo"],
    "Power Level": {"type": "number", "min": 1, "max": 100}
}

```

```

token_data = generate_random_collection(100, trait_options)

```

```

generator = MetadataGenerator(
    collection_name="Cool Characters",
    description_template="Cool Character #{token_id} from the
    genesis collection.",
    base_image_uri="ipfs://QmXxx..."
)

```

```

metadata_list = generator.generate_collection(token_data)
generator.save_to_files(metadata_list, "./metadata")

```

SECTION 3: HOW IPFS WORKS

IPFS (InterPlanetary File System) is a peer-to-peer network for storing and sharing files. Understanding its architecture helps you use it correctly.

Content Addressing

Traditional web: Files identified by location (URL). `https://server.com/image.png` points to a server. The server controls what that URL returns.

IPFS: Files identified by content hash (CID). `ipfs://QmXnYz...` is a cryptographic hash of the file content. The hash guarantees the content -- if the content changes, the hash changes.

This means:

- Same content always produces same CID
- CID verifies content integrity automatically
- Content cannot be modified without changing CID
- Anyone with the CID can verify they have the correct file

CID Versions

CIDv0: Starts with "Qm", uses Base58 encoding. Legacy format.

Example:

`QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG`

CIDv1: More flexible, supports multiple hash algorithms and encodings.

Example:

`bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3ocltqy55fbzdi`

Both work. Most NFT projects use CIDv0 for familiarity.

How Content Distributes

When you add a file to IPFS:

1. File is split into chunks
2. Each chunk is hashed
3. Chunks form a Merkle DAG (directed acyclic graph)
4. Root hash becomes the CID

When someone requests the CID:

1. Their IPFS node asks the network "who has this CID?"
2. Nodes that have the content respond
3. Content transfers peer-to-peer

4. Receiving node verifies hash matches CID
5. Receiving node can now serve this content to others

The Pinning Problem

IPFS nodes don't store everything forever. By default, content is cached temporarily then garbage collected. If nobody pins the content, it eventually disappears.

Pinning means telling a node "keep this content permanently." Your local node, a pinning service, or multiple services can pin content.

For NFTs, this is critical. If nobody pins your images, your NFTs become pointers to nothing.

SECTION 4: IPFS PINNING SERVICES

Several services provide reliable IPFS pinning:

Pinata

The most popular NFT pinning service:

import requests

```
class PinataClient:
```

```
    def __init__(self, api_key, secret_key):
        self.api_key = api_key
        self.secret_key = secret_key
        self.base_url = "https://api.pinata.cloud"
```

```
    def get_headers(self):
        return {
            "pinata_api_key": self.api_key,
            "pinata_secret_api_key": self.secret_key
        }
```

```
    def pin_file(self, file_path, name=None):
        url = f'{self.base_url}/pinning/pinFileToIPFS'
```

```
with open(file_path, 'rb') as f:  
    files = {'file': f}
```

```
metadata = {"name": name or file_path}  
data = {"pinataMetadata": json.dumps(metadata)}
```

```
response = requests.post(  
    url,  
    files=files,  
    data=data,  
    headers=self.get_headers()  
)
```

```
return response.json()
```

```
def pin_json(self, json_data, name=None):  
    url = f'{self.base_url}/pinning/pinJSONToIPFS'
```

```
    body = {  
        "pinataContent": json_data,  
        "pinataMetadata": {"name": name or "metadata.json"}  
    }
```

```
    response = requests.post(  
        url,  
        json=body,  
        headers={**self.get_headers(), "Content-Type": "application/json"}  
    )
```

```
    return response.json()
```

```
def pin_directory(self, directory_path, name=None):  
    url = f'{self.base_url}/pinning/pinFileToIPFS'
```

```
    files = []  
    for root, dirs, filenames in os.walk(directory_path):  
        for filename in filenames:  
            filepath = os.path.join(root, filename)  
            relative_path = os.path.relpath(filepath, directory_path)
```

```
files.append(  
(file, (relative_path, open(filepath, 'rb'))))  
)
```

```
metadata = {"name": name or directory_path}  
data = {"pinataMetadata": json.dumps(metadata)}
```

```
response = requests.post(  
url,  
files=files,  
data=data,  
headers=self.get_headers()  
)
```

```
for _, (f) in files:  
f.close()
```

```
return response.json()
```

```
def unpin(self, cid):  
url = f'{self.base_url}/pinning/unpin/{cid}'
```

```
response = requests.delete(url, headers=self.get_headers())  
return response.status_code == 200
```

```
def list_pins(self, status="pinned"):  
url = f'{self.base_url}/data/pinList'  
params = {"status": status}
```

```
response = requests.get(url, params=params,  
headers=self.get_headers())  
return response.json()
```

```
pinata = PinataClient("your_api_key", "your_secret_key")
```

```
result = pinata.pin_file("./images/1.png", "NFT Image 1")  
print(f'CID: {result["IpfsHash"]}')  

```

```
metadata = {"name": "Test NFT", "description": "A test", "image":
```

```
f'ipfs://{result['IpfsHash']}"})
meta_result = pinata.pin_json(metadata, "metadata_1.json")
print(f'Metadata CID: {meta_result['IpfsHash']})
```

NFT.Storage

Free storage for NFTs (funded by Protocol Labs):

```
class NFTStorageClient:
    def __init__(self, api_token):
        self.api_token = api_token
        self.base_url = "https://api.nft.storage"

    def get_headers(self):
        return {
            "Authorization": f"Bearer {self.api_token}"
        }

    def upload_file(self, file_path):
        url = f"{self.base_url}/upload"

        with open(file_path, 'rb') as f:
            response = requests.post(
                url,
                data=f,
                headers={
                    **self.get_headers(),
                    "Content-Type": "application/octet-stream"
                }
            )

        return response.json()

    def upload_directory(self, directory_path):
        url = f"{self.base_url}/upload"

        files = []
        for root, dirs, filenames in os.walk(directory_path):
            for filename in filenames:
                filepath = os.path.join(root, filename)
```

```

relative_path = os.path.relpath(filepath, directory_path)
files.append(
('file', (relative_path, open(filepath, 'rb'))))
)

```

```

response = requests.post(url, files = files,
headers = self.get_headers())

```

```

for _, (_, f) in files:
f.close()

```

```

return response.json()

```

```

def store_metadata(self, metadata, image_cid):
metadata['image'] = f'ipfs://{image_cid}"

```

```

url = f'{self.base_url}/upload"

```

```

response = requests.post(
url,
json = metadata,
headers = {
**self.get_headers(),
"Content-Type": "application/json"
}
)

```

```

return response.json()

```

```

def check_status(self, cid):
url = f'{self.base_url}/check/{cid}"
response = requests.get(url, headers = self.get_headers())
return response.json()

```

SECTION 5: UPLOADING NFT COLLECTIONS

Complete workflow for uploading a collection:

```

import os

```

```

import json
import time

class CollectionUploader:
    def __init__(self, pinata_client):
        self.pinata = pinata_client

    def upload_collection(self, images_dir, metadata_dir,
collection_name):
        print("Uploading images directory...")
        images_result = self.pinata.pin_directory(images_dir,
f'{collection_name}_images')
        images_cid = images_result['IpfsHash']
        print(f'Images CID: {images_cid}')

        print("Updating metadata with image CIDs...")
        self.update_metadata_images(metadata_dir, images_cid)

        print("Uploading metadata directory...")
        metadata_result = self.pinata.pin_directory(metadata_dir,
f'{collection_name}_metadata')
        metadata_cid = metadata_result['IpfsHash']
        print(f'Metadata CID: {metadata_cid}')

        return {
            "images_cid": images_cid,
            "metadata_cid": metadata_cid,
            "base_uri": f'ipfs://{metadata_cid}/'
        }

    def update_metadata_images(self, metadata_dir, images_cid):
        for filename in os.listdir(metadata_dir):
            if filename.endswith('.json'):
                filepath = os.path.join(metadata_dir, filename)

                with open(filepath, 'r') as f:
                    metadata = json.load(f)

                token_id = filename.replace('.json', '')
                metadata['image'] = f'ipfs://{images_cid}/{token_id}.png"

```

```

with open(filepath, 'w') as f:
    json.dump(metadata, f, indent = 2)

def verify_upload(self, cid, expected_files):
    gateway = f"https://gateway.pinata.cloud/ipfs/{cid}"

    print(f"Verifying upload at {gateway}")

    for filename in expected_files[:5]:
        url = f"{gateway}/{filename}"
        try:
            response = requests.head(url, timeout = 30)
            status = "OK" if response.status_code == 200 else "FAILED"
            print(f" {filename}: {status}")
        except Exception as e:
            print(f" {filename}: ERROR - {e}")

def upload_single_nft(self, image_path, metadata):
    image_result = self.pinata.pin_file(image_path)
    image_cid = image_result['IpfsHash']

    metadata['image'] = f'ipfs://{image_cid}'

    metadata_result = self.pinata.pin_json(metadata)
    metadata_cid = metadata_result['IpfsHash']

    return {
        "image_cid": image_cid,
        "metadata_cid": metadata_cid,
        "token_uri": f'ipfs://{metadata_cid}'
    }

```

SECTION 6: ON-CHAIN METADATA

For maximum trustlessness, store metadata directly on-chain:

```

contract OnChainNFT {
    struct TokenData {

```



```
string name;  
string description;  
string svgData;  
string[] traitTypes;  
string[] traitValues;  
}
```

```
mapping(uint256 => TokenData) private _tokenData;  
mapping(uint256 => address) private _owners;  
mapping(address => uint256) private _balances;
```

```
uint256 private _currentTokenId;  
address public owner;
```

```
constructor() {  
    owner = msg.sender;  
}
```

```
function mint(  
    address to,  
    string memory name,  
    string memory description,  
    string memory svgData,  
    string[] memory traitTypes,  
    string[] memory traitValues  
) external returns (uint256) {  
    require(msg.sender == owner);  
    require(traitTypes.length == traitValues.length);
```

```
    uint256 tokenId = _currentTokenId;  
    _currentTokenId ++;
```

```
    _owners[tokenId] = to;  
    _balances[to] ++;
```

```
    _tokenData[tokenId] = TokenData({  
        name: name,  
        description: description,  
        svgData: svgData,  
        traitTypes: traitTypes,
```

```
traitValues: traitValues
});
```

```
return tokenId;
}
```

```
function tokenId(uint256 tokenId) public view returns (string
memory) {
    require(_owners[tokenId] != address(0));
```

```
TokenData memory data = _tokenData[tokenId];
```

```
string memory attributes = _buildAttributes(data.traitTypes,
data.traitValues);
```

```
string memory json = string(abi.encodePacked(
'{"name":', data.name,
'", "description":', data.description,
'", "image": "data:image/svg+xml;base64,',
_encode(bytes(data.svgData)),
'", "attributes":', attributes,
'}'
));
```

```
return string(abi.encodePacked(
'data:application/json;base64,',
_encode(bytes(json))
));
}
```

```
function _buildAttributes(
string[] memory types,
string[] memory values
) internal pure returns (string memory) {
    if (types.length == 0) return "[]";
```

```
string memory result = "[";
```

```
for (uint256 i = 0; i < types.length; i++) {
    if (i > 0) result = string(abi.encodePacked(result, ","));
```

```

result = string(abi.encodePacked(
result,
'{"trait_type":', types[i],
'', "value":', values[i], ''}
));
}

return string(abi.encodePacked(result, ""]);
}

```

```

function _encode(bytes memory data) internal pure returns (string
memory) {
string memory TABLE =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345

if (data.length == 0) return "";

uint256 encodedLen = 4 * ((data.length + 2) / 3);
bytes memory result = new bytes(encodedLen);

bytes memory table = bytes(TABLE);

uint256 i = 0;
uint256 j = 0;

while (i < data.length) {
uint256 a = uint8(data[i + ]);
uint256 b = i < data.length ? uint8(data[i + ]) : 0;
uint256 c = i < data.length ? uint8(data[i + ]) : 0;

uint256 triple = (a << 16) + (b << 8) + c;

result[j + ] = table[(triple >> 18) & 0x3F];
result[j + ] = table[(triple >> 12) & 0x3F];
result[j + ] = table[(triple >> 6) & 0x3F];
result[j + ] = table[triple & 0x3F];
}

uint256 mod = data.length % 3;
if (mod > 0) {

```

```

result[encodedLen - 1] = "=";
if (mod == 1) result[encodedLen - 2] = "=";
}

```

```

return string(result);
}

```

```

function ownerOf(uint256 tokenId) public view returns (address) {
    address tokenOwner = _owners[tokenId];
    require(tokenOwner != address(0));
    return tokenOwner;
}

```

```

function balanceOf(address account) public view returns (uint256)
{
    require(account != address(0));
    return _balances[account];
}
}

```

On-chain SVG generation:

```

contract GenerativeSVGNFT {
    mapping(uint256 => uint256) private _seeds;
    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;

    uint256 private _currentTokenId;
    address public owner;

    string[] private backgrounds = ["#FF6B6B", "#4ECDC4",
    "#45B7D1", "#96CEB4", "#FFEAA7"];
    string[] private shapes = ["circle", "rect", "polygon"];

    constructor() {
        owner = msg.sender;
    }

    function mint(address to) external returns (uint256) {
        uint256 tokenId = _currentTokenId;

```

```
_currentTokenId + +;
```

```
_owners[tokenId] = to;
```

```
_balances[to] + +;
```

```
_seeds[tokenId] = uint256(keccak256(abi.encodePacked(  
block.timestamp,  
block.prevrandao,  
tokenId,  
to  
)));
```

```
return tokenId;
```

```
}
```

```
function tokenURI(uint256 tokenId) public view returns (string  
memory) {
```

```
require(_owners[tokenId] != address(0));
```

```
uint256 seed = _seeds[tokenId];
```

```
string memory svg = _generateSVG(seed);
```

```
string memory attributes = _generateAttributes(seed);
```

```
string memory json = string(abi.encodePacked(  
'{"name":"Generative #', _toString(tokenId),  
"', "description":"A fully on-chain generative artwork",',  
"'image":"data:image/svg+xml;base64,', _encode(bytes(svg)),  
"', "attributes":', attributes, '}'  
));
```

```
return string(abi.encodePacked(  
'data:application/json;base64',  
_encode(bytes(json))  
));
```

```
}
```

```
function _generateSVG(uint256 seed) internal view returns (string  
memory) {
```

```
string memory bgColor = backgrounds[seed %  
backgrounds.length];
```

```

string memory svg = string(abi.encodePacked(
'<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 500
500">',
'<rect width="500" height="500" fill="", bgColor, ""/>'
));

```

```

uint256 shapeCount = 3 + (seed % 5);

```

```

for (uint256 i = 0; i < shapeCount; i++) {
    uint256 shapeSeed = uint256(keccak256(abi.encodePacked(seed,
i)));
    svg = string(abi.encodePacked(svg, _generateShape(shapeSeed)));
}

```

```

svg = string(abi.encodePacked(svg, '</svg>'));

```

```

return svg;
}

```

```

function _generateShape(uint256 seed) internal pure returns (string
memory) {

```

```

    uint256 x = 50 + (seed % 400);
    uint256 y = 50 + ((seed >> 8) % 400);
    uint256 size = 20 + ((seed >> 16) % 80);

```

```

    uint256 r = (seed >> 24) % 256;
    uint256 g = (seed >> 32) % 256;
    uint256 b = (seed >> 40) % 256;

```

```

    return string(abi.encodePacked(
'<circle cx="", _toString(x),
"" cy="", _toString(y),
"" r="", _toString(size),
"" fill="rgb(', _toString(r), ',', _toString(g), ',', _toString(b),
')" opacity="0.7"/>'
));
}

```

```

function _generateAttributes(uint256 seed) internal view returns

```

```
(string memory) {
    string memory bgColor = backgrounds[seed %
backgrounds.length];
    uint256 shapeCount = 3 + (seed % 5);

    return string(abi.encodePacked(
        '{"trait_type":"Background","value":', bgColor,
        '},{\"trait_type\":\"Shape Count\",\"display_type\":\"number\",\"value\":',
        _toString(shapeCount),
        '}]'
    ));
}
```

```
function _toString(uint256 value) internal pure returns (string
memory) {
    if (value == 0) return "0";
    uint256 temp = value;
    uint256 digits;
    while (temp != 0) {
        digits++;
        temp /= 10;
    }
    bytes memory buffer = new bytes(digits);
    while (value != 0) {
        digits--;
        buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
        value /= 10;
    }
    return string(buffer);
}
```

```
function _encode(bytes memory data) internal pure returns (string
memory) {
    string memory TABLE =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    if (data.length == 0) return "";

    uint256 encodedLen = 4 * ((data.length + 2) / 3);
    bytes memory result = new bytes(encodedLen);
```

```

bytes memory table = bytes(TABLE);

uint256 i = 0;
uint256 j = 0;

while (i < data.length) {
    uint256 a = uint8(data[i + +]);
    uint256 b = i < data.length ? uint8(data[i + +]) : 0;
    uint256 c = i < data.length ? uint8(data[i + +]) : 0;

    uint256 triple = (a << 16) + (b << 8) + c;

    result[j + +] = table[(triple >> 18) & 0x3F];
    result[j + +] = table[(triple >> 12) & 0x3F];
    result[j + +] = table[(triple >> 6) & 0x3F];
    result[j + +] = table[triple & 0x3F];
}

uint256 mod = data.length % 3;
if (mod > 0) {
    result[encodedLen - 1] = "=";
    if (mod == 1) result[encodedLen - 2] = "=";
}

return string(result);
}

function ownerOf(uint256 tokenId) public view returns (address) {
    return _owners[tokenId];
}

function balanceOf(address account) public view returns (uint256)
{
    return _balances[account];
}
}

```

SECTION 7: METADATA IMMUTABILITY

Ensuring metadata cannot change after minting:

Frozen Base URI

```
contract FrozenMetadataNFT {
    string private _baseURI;
    bool public metadataFrozen;

    address public owner;
    uint256 private _currentTokenId;
    mapping(uint256 => address) private _owners;

    event MetadataFrozen(string baseURI);

    constructor(string memory initialBaseURI) {
        owner = msg.sender;
        _baseURI = initialBaseURI;
    }

    function setBaseURI(string memory newBaseURI) external {
        require(msg.sender == owner);
        require(!metadataFrozen);
        _baseURI = newBaseURI;
    }

    function freezeMetadata() external {
        require(msg.sender == owner);
        require(!metadataFrozen);

        metadataFrozen = true;
        emit MetadataFrozen(_baseURI);
    }

    function tokenURI(uint256 tokenId) public view returns (string
memory) {
        require(_owners[tokenId] != address(0));
        return string(abi.encodePacked(_baseURI, _toString(tokenId),
".json"));
    }
}
```

```

function _toString(uint256 value) internal pure returns (string
memory) {
    if (value == 0) return "0";
    uint256 temp = value;
    uint256 digits;
    while (temp != 0) {
        digits++;
        temp /= 10;
    }
    bytes memory buffer = new bytes(digits);
    while (value != 0) {
        digits--;
        buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
        value /= 10;
    }
    return string(buffer);
}
}

```

Per-Token URI Storage

```

contract IndividualMetadataNFT {
    mapping(uint256 => string) private _tokenURIs;
    mapping(uint256 => bool) private _tokenURIFrozen;
    mapping(uint256 => address) private _owners;

    address public owner;
    uint256 private _currentTokenId;

    event TokenURIFrozen(uint256 indexed tokenId, string uri);

    constructor() {
        owner = msg.sender;
    }

    function setTokenURI(uint256 tokenId, string memory uri) external
    {
        require(msg.sender == owner);
        require(!_tokenURIFrozen[tokenId]);
        _tokenURIs[tokenId] = uri;
    }
}

```

```
}
```

```
function freezeTokenURI(uint256 tokenId) external {  
    require(msg.sender == owner);  
    require(bytes(_tokenURIs[tokenId]).length > 0);  
    require(!_tokenURIFrozen[tokenId]);
```

```
    _tokenURIFrozen[tokenId] = true;  
    emit TokenURIFrozen(tokenId, _tokenURIs[tokenId]);  
}
```

```
function batchFreezeTokenURIs(uint256[] calldata tokenIds)  
external {  
    require(msg.sender == owner);
```

```
    for (uint256 i = 0; i < tokenIds.length; i++) {  
        uint256 tokenId = tokenIds[i];  
        if (!_tokenURIFrozen[tokenId] &&  
            bytes(_tokenURIs[tokenId]).length > 0) {  
            _tokenURIFrozen[tokenId] = true;  
            emit TokenURIFrozen(tokenId, _tokenURIs[tokenId]);  
        }  
    }  
}
```

```
function tokenURI(uint256 tokenId) public view returns (string  
memory) {  
    require(_owners[tokenId] != address(0));  
    return _tokenURIs[tokenId];  
}
```

```
function isTokenURIFrozen(uint256 tokenId) public view returns  
(bool) {  
    return _tokenURIFrozen[tokenId];  
}  
}
```

SECTION 8: REVEAL MECHANICS

Many collections launch with hidden metadata, then reveal:

```
contract RevealableNFT {
    string private _unrevealedURI;
    string private _baseURI;
    bool public revealed;

    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;

    address public owner;
    uint256 public totalSupply;
    uint256 public maxSupply;

    event Revealed(string baseURI);

    constructor(
        string memory unrevealedURI,
        uint256 supply
    ) {
        owner = msg.sender;
        _unrevealedURI = unrevealedURI;
        maxSupply = supply;
    }

    function reveal(string memory baseURI) external {
        require(msg.sender == owner);
        require(!revealed);

        _baseURI = baseURI;
        revealed = true;

        emit Revealed(baseURI);
    }

    function tokenURI(uint256 tokenId) public view returns (string
memory) {
        require(_owners[tokenId] != address(0));

        if (!revealed) {
```

```

return _unrevealedURI;
}

return string(abi.encodePacked(_baseURI, _toString(tokenId),
".json"));
}

function _toString(uint256 value) internal pure returns (string
memory) {
    if (value == 0) return "0";
    uint256 temp = value;
    uint256 digits;
    while (temp != 0) {
        digits++;
        temp /= 10;
    }
    bytes memory buffer = new bytes(digits);
    while (value != 0) {
        digits--;
        buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
        value /= 10;
    }
    return string(buffer);
}
}

```

Randomized Reveal

To prevent gaming the reveal by knowing which token ID gets which metadata:

```

contract RandomizedRevealNFT {
    string private _unrevealedURI;
    string private _baseURI;

    bool public revealed;
    uint256 public randomOffset;

    mapping(uint256 => address) private _owners;
    address public owner;
}

```

```
uint256 public totalSupply;
uint256 public maxSupply;
```

```
constructor(string memory unrevealedURI, uint256 supply) {
    owner = msg.sender;
    _unrevealedURI = unrevealedURI;
    maxSupply = supply;
}
```

```
function reveal(string memory baseURI, uint256 offset) external {
    require(msg.sender == owner);
    require(!revealed);
    require(offset < maxSupply);
```

```
    _baseURI = baseURI;
    randomOffset = offset;
    revealed = true;
}
```

```
function tokenURI(uint256 tokenId) public view returns (string
memory) {
    require(_owners[tokenId] != address(0));
```

```
    if (!revealed) {
        return _unrevealedURI;
    }
```

```
    uint256 metadataId = (tokenId + randomOffset) % maxSupply;
    return string(abi.encodePacked(_baseURI, _toString(metadataId),
".json"));
}
```

```
function _toString(uint256 value) internal pure returns (string
memory) {
    if (value == 0) return "0";
    uint256 temp = value;
    uint256 digits;
    while (temp != 0) {
        digits++;
        temp /= 10;
    }
```

```

}
bytes memory buffer = new bytes(digits);
while (value != 0) {
  digits -= 1;
  buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
  value /= 10;
}
return string(buffer);
}
}

```

For provably fair randomness, use Chainlink VRF to generate the offset.

SECTION 9: FETCHING AND DISPLAYING METADATA

Client-side metadata handling:

```

import { ethers } from 'ethers'

class MetadataFetcher {
  constructor(ipfsGateways = null) {
    this.gateways = ipfsGateways || [
      'https://ipfs.io/ipfs/',
      'https://gateway.pinata.cloud/ipfs/',
      'https://cloudflare-ipfs.com/ipfs/',
      'https://dweb.link/ipfs/'
    ]
  }

  async fetchMetadata(tokenURI) {
    const url = this.resolveURI(tokenURI)
    const response = await this.fetchWithFallback(url)
    return response
  }

  resolveURI(uri) {
    if (uri.startsWith('ipfs://')) {
      const hash = uri.replace('ipfs://', '')

```

```
return { type: 'ipfs', hash, url: `${this.gateways[0]}${hash}` }  
}
```

```
if (uri.startsWith('ar://')) {  
  const hash = uri.replace('ar://', '')  
  return { type: 'arweave', hash, url: `https://arweave.net/${hash}` }  
}
```

```
if (uri.startsWith('data:application/json;base64,')) {  
  const base64 = uri.replace('data:application/json;base64,', '')  
  const json = atob(base64)  
  return { type: 'data', data: JSON.parse(json) }  
}
```

```
if (uri.startsWith('data:application/json,')) {  
  const json = uri.replace('data:application/json,', '')  
  return { type: 'data', data: JSON.parse(decodeURIComponent(json)) }  
}  
}
```

```
return { type: 'http', url: uri }  
}
```

```
async fetchWithFallback(resolved) {  
  if (resolved.type === 'data') {  
    return resolved.data  
  }
```

```
  if (resolved.type === 'ipfs') {  
    for (const gateway of this.gateways) {  
      try {  
        const response = await fetch(`${gateway}${resolved.hash}`, {  
          timeout: 10000  
        })  
        if (response.ok) {  
          return response.json()  
        }  
      } catch (e) {  
        continue  
      }  
    }  
  }
```



```
}  
throw new Error('All IPFS gateways failed')  
}
```

```
const response = await fetch(resolved.url)  
return response.json()  
}
```

```
async resolveImage(imageURI) {  
  if (imageURI.startsWith('ipfs://')) {  
    const hash = imageURI.replace('ipfs://', '')  
    return `${this.gateways[0]}${hash}`  
  }
```

```
  if (imageURI.startsWith('ar://')) {  
    const hash = imageURI.replace('ar://', '')  
    return `https://arweave.net/${hash}`  
  }
```

```
  if (imageURI.startsWith('data:')) {  
    return imageURI  
  }
```

```
  return imageURI  
}  
}
```

```
class NFTDisplay {  
  constructor(contractAddress, provider) {  
    this.contract = new ethers.Contract(  
      contractAddress,  
      ['function tokenURI(uint256) view returns (string)'],  
      provider  
    )  
    this.metadataFetcher = new MetadataFetcher()  
  }
```

```
  async getFullNFTData(tokenId) {  
    const tokenURI = await this.contract.tokenURI(tokenId)  
    const metadata = await
```

```

this.metadataFetcher.fetchMetadata(tokenURI)

const imageUrl = await
this.metadataFetcher.resolveImage(metadata.image)

let animationUrl = null
if (metadata.animation_url) {
  animationUrl = await
this.metadataFetcher.resolveImage(metadata.animation_url)
}

return {
  tokenId,
  tokenURI,
  name: metadata.name,
  description: metadata.description,
  imageUrl,
  animationUrl,
  attributes: metadata.attributes || [],
  externalUrl: metadata.external_url,
  backgroundColor: metadata.background_color,
  rawMetadata: metadata
}
}

async getCollectionMetadata(tokenIds) {
  const results = await Promise.all(
    tokenIds.map(id => this.getFullNFTData(id).catch(e => ({
      tokenId: id, error: e.message })))
  )
  return results
}
}

```

SECTION 10: METADATA BEST PRACTICES

Lessons from thousands of NFT launches:

Use IPFS or Arweave

Never use HTTP URLs for important assets. Centralized servers fail, change, or get abandoned. Use IPFS with reliable pinning or Arweave for permanence.

Pin with Multiple Services

Don't rely on one pinning service. Pin your content with 2-3 services. If one fails, others maintain availability.

Verify Before Launch

Test every metadata file and image:

- Verify all JSON is valid
- Check all image URLs resolve
- Confirm attributes are correctly formatted
- Test on OpenSea testnet before mainnet

Use Consistent Naming

Token 1 should have metadata at /1.json and image at /1.png. Consistent, predictable naming prevents errors.

Include All Standard Fields

Even optional fields improve marketplace display:

- Always include name and description
- Always include image
- Include external_url for project discovery
- Include attributes for filtering and rarity

Optimize Images

Large images slow loading and waste storage:

- Use 1000x1000 or 2000x2000 maximum
- Compress without visible quality loss
- Consider WebP for smaller files
- Provide thumbnails for galleries

Plan for Reveals

If doing a reveal:

- Upload all revealed metadata before launch
- Test the reveal mechanism on testnet
- Consider using Chainlink VRF for randomness
- Communicate timeline to community

Document Your Metadata

Create a public spec:

- List all trait types and possible values
- Explain any special attributes
- Document rarity distribution
- Provide examples of metadata structure

Metadata makes NFTs meaningful. The token ID is just a number -- metadata gives it name, image, attributes, and context. Invest in solid metadata infrastructure and your NFTs will display correctly, maintain value, and survive whatever happens to any single service provider.

CHAPTER 5: NFT MARKETPLACES INTEGRATION

Marketplaces are where NFTs find buyers. OpenSea, Blur, Rarible, Magic Eden, and others provide liquidity that makes NFTs valuable. Without marketplaces, selling an NFT means finding buyers yourself -- difficult and inefficient.

This chapter shows how to integrate with major marketplaces programmatically. List NFTs for sale, execute purchases, query listings, and handle royalties. You'll understand the protocols that power NFT trading and how to build applications that interact with them.

SECTION 1: MARKETPLACE LANDSCAPE

The NFT marketplace ecosystem has evolved rapidly:

OpenSea dominated early, pioneering the aggregator model where any ERC-721 or ERC-1155 automatically appears. Their Seaport

protocol now powers many marketplaces.

Blur challenged OpenSea with zero fees, optional royalties, and professional trading features. Traders migrated for better execution.

Rarible pioneered creator royalties and community governance. Their protocol remains widely used.

LooksRare and X2Y2 offered token incentives to attract traders. Volume flowed but often proved to be wash trading.

Magic Eden leads Solana and expanded to Ethereum. Strong in gaming NFTs.

Foundation focuses on curated art. Higher quality, smaller volume.

Each marketplace has different fees, royalty policies, and protocols. Professional applications aggregate across marketplaces to find best prices.

SECTION 2: OPENSEA SEAPORT PROTOCOL

Seaport is OpenSea's open-source protocol for NFT trading. Understanding Seaport means understanding modern NFT trading.

Core Concepts

Orders: Signed messages describing what a trader offers and wants. No on-chain transaction until execution.

Offer Items: What the order creator gives (NFTs or tokens).

Consideration Items: What the order creator wants to receive.

Fulfillment: Matching offer items to consideration items to execute trades.

Order Structure

```

struct OrderParameters {
    address offerer;
    address zone;
    OfferItem[] offer;
    ConsiderationItem[] consideration;
    OrderType orderType;
    uint256 startTime;
    uint256 endTime;
    bytes32 zoneHash;
    uint256 salt;
    bytes32 conduitKey;
    uint256 totalOriginalConsiderationItems;
}

```

```

struct OfferItem {
    ItemType itemType;
    address token;
    uint256 identifierOrCriteria;
    uint256 startAmount;
    uint256 endAmount;
}

```

```

struct ConsiderationItem {
    ItemType itemType;
    address token;
    uint256 identifierOrCriteria;
    uint256 startAmount;
    uint256 endAmount;
    address payable recipient;
}

```

Item types:

- 0: Native token (ETH)
- 1: ERC-20
- 2: ERC-721
- 3: ERC-1155
- 4: ERC-721 with criteria
- 5: ERC-1155 with criteria

Creating a Listing

```

import { ethers } from 'ethers'
import { Seaport } from '@opensea/seaport-js'

class SeaportClient {
  constructor(provider, seaportAddress) {
    this.provider = provider
    this.seaportAddress = seaportAddress
    this.seaport = new Seaport(provider)
  }

  async createListing(signer, params) {
    const {
      tokenAddress,
      tokenId,
      tokenType,
      price,
      currency,
      expirationTime,
      royaltyRecipient,
      royaltyBps
    } = params

    const offerer = await signer.getAddress()

    const offerItem = {
      itemType: tokenType === 'ERC721' ? 2 : 3,
      token: tokenAddress,
      identifier: tokenId,
      amount: '1'
    }

    const considerationItems = []

    let sellerAmount = price
    if (royaltyRecipient && royaltyBps > 0) {
      const royaltyAmount = (BigInt(price) * BigInt(royaltyBps)) /
      10000n
      sellerAmount = (BigInt(price) - royaltyAmount).toString()
    }
  }
}

```

```

considerationItems.push({
  itemType: currency === ethers.ZeroAddress ? 0 : 1,
  token: currency,
  identifier: '0',
  amount: royaltyAmount.toString(),
  recipient: royaltyRecipient
})
}

```

```

considerationItems.unshift({
  itemType: currency === ethers.ZeroAddress ? 0 : 1,
  token: currency,
  identifier: '0',
  amount: sellerAmount,
  recipient: offerer
})

```

```

const { executeAllActions } = await this.seaport.createOrder({
  offer: [offerItem],
  consideration: considerationItems,
  endTime: Math.floor(expirationTime / 1000).toString(),
  zone: ethers.ZeroAddress,
  restrictedByZone: false
}, offerer)

```

```

const order = await executeAllActions()
return order
}

```

```

async fulfillListing(signer, order) {
  const { executeAllActions } = await this.seaport.fulfillOrder({
    order,
    accountAddress: await signer.getAddress()
  })
}

```

```

const transaction = await executeAllActions()
return transaction
}

```

```

async createOffer(signer, params) {

```



```
const {  
  tokenAddress,  
  tokenId,  
  tokenType,  
  offerAmount,  
  currency,  
  expirationTime  
} = params
```

```
const offerer = await signer.getAddress()
```

```
const offerItem = {  
  itemType: currency === ethers.ZeroAddress ? 0 : 1,  
  token: currency,  
  identifier: '0',  
  amount: offerAmount  
}
```

```
const considerationItem = {  
  itemType: tokenType === 'ERC721' ? 2 : 3,  
  token: tokenAddress,  
  identifier: tokenId,  
  amount: '1',  
  recipient: offerer  
}
```

```
const { executeAllActions } = await this.seaport.createOrder({  
  offer: [offerItem],  
  consideration: [considerationItem],  
  endTime: Math.floor(expirationTime / 1000).toString()  
}, offerer)
```

```
const order = await executeAllActions()  
return order  
}
```

```
async cancelOrder(signer, orderComponents) {  
  const transaction = await this.seaport.cancelOrders(  
    [orderComponents],  
    await signer.getAddress()  
  )  
}
```

```

)
return transaction.wait()
}

async cancelAllOrders(signer) {
const transaction = await this.seaport.incrementCounter(
await signer.getAddress()
)
return transaction.wait()
}
}

```

SECTION 3: OPENSEA API INTEGRATION

OpenSea's API provides listing data, collection info, and order management:

```

class OpenSeaAPI {
  constructor(apiKey, chain = 'ethereum') {
    this.apiKey = apiKey
    this.chain = chain
    this.baseUrl = 'https://api.opensea.io/api/v2'
  }

  getHeaders() {
    return {
      'X-API-KEY': this.apiKey,
      'Accept': 'application/json'
    }
  }

  async getCollection(collectionSlug) {
    const response = await fetch(
      `${this.baseUrl}/collections/${collectionSlug}`,
      { headers: this.getHeaders() }
    )
    return response.json()
  }
}

```

```

async getCollectionStats(collectionSlug) {
  const response = await fetch(
    `${this.baseUrl}/collections/${collectionSlug}/stats`,
    { headers: this.getHeaders() }
  )
  return response.json()
}

```

```

async getNFT(contractAddress, tokenId) {
  const response = await fetch(
    `${this.baseUrl}/chain/${this.chain}/contract/
    ${contractAddress}/nfts/${tokenId}`,
    { headers: this.getHeaders() }
  )
  return response.json()
}

```

```

async getNFTsByContract(contractAddress, limit = 50, next =
null) {
  let url = `${this.baseUrl}/chain/${this.chain}/contract/
  ${contractAddress}/nfts?limit=${limit}`
  if (next) url += `&next=${next}`

```

```

const response = await fetch(url, { headers: this.getHeaders() })
return response.json()
}

```

```

async getNFTsByAccount(address, limit = 50, next = null) {
  let url = `${this.baseUrl}/chain/${this.chain}/account/
  ${address}/nfts?limit=${limit}`
  if (next) url += `&next=${next}`

```

```

const response = await fetch(url, { headers: this.getHeaders() })
return response.json()
}

```

```

async getListings(collectionSlug, limit = 50, next = null) {
  let url = `${this.baseUrl}/listings/collection/${collectionSlug}/
  all?limit=${limit}`
  if (next) url += `&next=${next}`

```

```
const response = await fetch(url, { headers: this.getHeaders() })
return response.json()
}
```

```
async getBestListing(collectionSlug, tokenId) {
const response = await fetch(
`${this.baseUrl}/listings/collection/${collectionSlug}/nfts/
${tokenId}/best`,
{ headers: this.getHeaders() }
)
return response.json()
}
```

```
async getOffers(collectionSlug, limit = 50) {
const response = await fetch(
`${this.baseUrl}/offers/collection/${collectionSlug}?limit =
${limit}`,
{ headers: this.getHeaders() }
)
return response.json()
}
```

```
async getBestOffer(collectionSlug, tokenId) {
const response = await fetch(
`${this.baseUrl}/offers/collection/${collectionSlug}/nfts/
${tokenId}/best`,
{ headers: this.getHeaders() }
)
return response.json()
}
```

```
async getCollectionOffers(collectionSlug) {
const response = await fetch(
`${this.baseUrl}/offers/collection/${collectionSlug}/all`,
{ headers: this.getHeaders() }
)
return response.json()
}
```

```

    async refreshMetadata(contractAddress, tokenId) {
      const response = await fetch(
        `${this.baseUrl}/chain/${this.chain}/contract/
        ${contractAddress}/nfts/${tokenId}/refresh`,
        {
          method: 'POST',
          headers: this.getHeaders()
        }
      )
      return response.status === 200
    }
  }

  async function monitorCollectionActivity(api, collectionSlug) {
    const stats = await api.getCollectionStats(collectionSlug)

    console.log(`Collection: ${collectionSlug}`)
    console.log(`Floor Price: ${stats.total.floor_price} ETH`)
    console.log(`Total Volume: ${stats.total.volume} ETH`)
    console.log(`Total Sales: ${stats.total.sales}`)
    console.log(`Num Owners: ${stats.total.num_owners}`)
    console.log(`Total Supply: ${stats.total.supply}`)

    const listings = await api.getListings(collectionSlug, 10)

    console.log(`\nCheapest Listings:`)
    for (const listing of listings.listings.slice(0, 5)) {
      const price = listing.price.current.value / 1e18
      console.log(`Token
      ${listing.protocol_data.parameters.offer[0].identifierOrCriteria}:
      ${price} ETH`)
    }
  }
}

```

SECTION 4: BLUR INTEGRATION

Blur focuses on professional traders with advanced features:

```

class BlurAPI {

```

```
constructor(apiKey) {  
  this.apiKey = apiKey  
  this.baseUrl = 'https://api.blur.io/v1'  
}
```

```
getHeaders() {  
  return {  
    'Authorization': `Bearer ${this.apiKey}`,  
    'Content-Type': 'application/json'  
  }  
}
```

```
async getCollection(contractAddress) {  
  const response = await fetch(  
    `${this.baseUrl}/collections/${contractAddress}`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getCollectionStats(contractAddress) {  
  const response = await fetch(  
    `${this.baseUrl}/collections/${contractAddress}/stats`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getListings(contractAddress, filters = {}) {  
  const params = new URLSearchParams(filters)  
  const response = await fetch(  
    `${this.baseUrl}/collections/${contractAddress}/tokens?  
${params}`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getTokenActivity(contractAddress, tokenId) {  
  const response = await fetch(  
    `${this.baseUrl}/tokens/${tokenId}/activity`  
  )  
  return response.json()  
}
```

```
`${this.baseUrl}/tokens/${contractAddress}/${tokenId}/activity`,  
{ headers: this.getHeaders() }  
)  
return response.json()  
}
```

```
async getBids(contractAddress) {  
  const response = await fetch(  
    `${this.baseUrl}/collections/${contractAddress}/bids`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getUserBids(userAddress) {  
  const response = await fetch(  
    `${this.baseUrl}/users/${userAddress}/bids`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getUserListings(userAddress) {  
  const response = await fetch(  
    `${this.baseUrl}/users/${userAddress}/listings`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getFloorAsk(contractAddress) {  
  const response = await fetch(  
    `${this.baseUrl}/collections/${contractAddress}/floor-ask`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}  
}
```

SECTION 5: AGGREGATOR INTEGRATION

Aggregators like Reservoir combine listings from multiple marketplaces:

```
class ReservoirAPI {  
  constructor(apiKey, baseUrl = 'https://api.reservoir.tools') {  
    this.apiKey = apiKey  
    this.baseUrl = baseUrl  
  }
```

```
  getHeaders() {  
    return {  
      'x-api-key': this.apiKey,  
      'Accept': 'application/json'  
    }  
  }
```

```
  async getTokens(params) {  
    const queryParams = new URLSearchParams(params)  
    const response = await fetch(  
      `${this.baseUrl}/tokens/v6?${queryParams}`,  
      { headers: this.getHeaders() }  
    )  
    return response.json()  
  }
```

```
  async getFloorPrice(collection) {  
    const response = await fetch(  
      `${this.baseUrl}/collections/v6?id=${collection}`,  
      { headers: this.getHeaders() }  
    )  
    const data = await response.json()  
    return data.collections[0]?.floorAsk?.price?.amount?.native  
  }
```

```
  async getListings(params) {  
    const queryParams = new URLSearchParams(params)  
    const response = await fetch(  
      `${this.baseUrl}/orders/asks/v5?${queryParams}`,
```



```
{ headers: this.getHeaders() }  
)  
return response.json()  
}
```

```
async getBids(params) {  
  const queryParams = new URLSearchParams(params)  
  const response = await fetch(  
    `${this.baseUrl}/orders/bids/v6?${queryParams}`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getSales(params) {  
  const queryParams = new URLSearchParams(params)  
  const response = await fetch(  
    `${this.baseUrl}/sales/v6?${queryParams}`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async getCollectionActivity(collection, limit = 20) {  
  const response = await fetch(  
    `${this.baseUrl}/collections/activity/v6?collection=${collection}  
&limit=${limit}`,  
    { headers: this.getHeaders() }  
  )  
  return response.json()  
}
```

```
async buyToken(params) {  
  const response = await fetch(  
    `${this.baseUrl}/execute/buy/v7`,  
    {  
      method: 'POST',  
      headers: {  
        ...this.getHeaders(),  
        'Content-Type': 'application/json'  
      }  
    }  
  )  
  return response.json()  
}
```

```
    },  
    body: JSON.stringify(params)  
  }  
)  
return response.json()  
}  
  
async listToken(params) {  
  const response = await fetch(  
    `${this.baseUrl}/execute/list/v5`,  
    {  
      method: 'POST',  
      headers: {  
        ...this.getHeaders(),  
        'Content-Type': 'application/json'  
      },  
      body: JSON.stringify(params)  
    }  
  )  
  return response.json()  
}
```

```
async placeBid(params) {  
  const response = await fetch(  
    `${this.baseUrl}/execute/bid/v5`,  
    {  
      method: 'POST',  
      headers: {  
        ...this.getHeaders(),  
        'Content-Type': 'application/json'  
      },  
      body: JSON.stringify(params)  
    }  
  )  
  return response.json()  
}  
}
```

```
async function findBestPrice(reservoir, collection, tokenId) {  
  const listings = await reservoir.getListings({
```

```
token: `${collection}:${tokenId}`,  
sortBy: 'price',  
limit: 10  
})
```

```
if (listings.orders.length === 0) {  
  return null  
}
```

```
const best = listings.orders[0]  
return {  
  price: best.price.amount.native,  
  marketplace: best.source.name,  
  orderId: best.id,  
  validUntil: best.validUntil  
}  
}
```

```
async function comparePricesAcrossMarketplaces(reservoir,  
collection) {  
  const tokens = await reservoir.getTokens({  
    collection: collection,  
    sortBy: 'floorAskPrice',  
    limit: 20  
  })
```

```
  const comparison = []
```

```
  for (const token of tokens.tokens) {  
    const listings = token.market?.floorAsk  
    if (listings) {  
      comparison.push({  
        tokenId: token.token.tokenId,  
        price: listings.price?.amount?.native,  
        marketplace: listings.source?.name,  
        lastSale: token.token.lastSale?.price?.amount?.native  
      })  
    }  
  }  
}
```

```
return comparison
}
```

SECTION 6: EXECUTING PURCHASES

Buying NFTs programmatically:

```
import { ethers } from 'ethers'

class NFTPurchaser {
  constructor(provider, reservoirApiKey) {
    this.provider = provider
    this.reservoir = new ReservoirAPI(reservoirApiKey)
  }

  async buyNFT(signer, collection, tokenId, maxPrice) {
    const listings = await this.reservoir.getListings({
      token: `${collection}:${tokenId}`,
      sortBy: 'price',
      limit: 1
    })

    if (listings.orders.length === 0) {
      throw new Error('No listings found')
    }

    const listing = listings.orders[0]
    const price = listing.price.amount.native

    if (price > maxPrice) {
      throw new Error(`Price ${price} exceeds max ${maxPrice}`)
    }

    const buyerAddress = await signer.getAddress()

    const buyData = await this.reservoir.buyToken({
      items: [{
        token: `${collection}:${tokenId}`,
        quantity: 1
```

```
    }],  
    taker: buyerAddress,  
    currency: ethers.ZeroAddress,  
    skipBalanceCheck: false  
  })
```

```
  if (buyData.steps) {  
    for (const step of buyData.steps) {  
      for (const item of step.items) {  
        if (item.status === 'incomplete') {  
          const tx = await signer.sendTransaction({  
            to: item.data.to,  
            data: item.data.data,  
            value: item.data.value || 0  
          })  
          await tx.wait()  
        }  
      }  
    }  
  }  
}
```

```
return { success: true, price }  
}
```

```
async sweepFloor(signer, collection, count, maxPricePerNFT) {  
  const tokens = await this.reservoir.getTokens({  
    collection: collection,  
    sortBy: 'floorAskPrice',  
    limit: count  
  })
```

```
  const items = []  
  let totalCost = 0
```

```
  for (const token of tokens.tokens) {  
    const price = token.market?.floorAsk?.price?.amount?.native  
    if (price && price <= maxPricePerNFT) {  
      items.push({  
        token: `${collection}:${token.token.tokenId}`,  
        quantity: 1
```

```

    })
    totalCost += price
  }
}

if (items.length === 0) {
  throw new Error('No tokens within price range')
}

const buyerAddress = await signer.getAddress()

const buyData = await this.reservoir.buyToken({
  items: items,
  taker: buyerAddress,
  currency: ethers.ZeroAddress
})

if (buyData.steps) {
  for (const step of buyData.steps) {
    for (const item of step.items) {
      if (item.status === 'incomplete') {
        const tx = await signer.sendTransaction({
          to: item.data.to,
          data: item.data.data,
          value: item.data.value || 0
        })
        await tx.wait()
      }
    }
  }
}

return { purchased: items.length, totalCost }
}

```

SECTION 7: LISTING NFTs FOR SALE

Creating listings across marketplaces:

```
class NFTLister {  
  constructor(provider, reservoirApiKey) {  
    this.provider = provider  
    this.reservoir = new ReservoirAPI(reservoirApiKey)  
  }
```

```
  async listNFT(signer, params) {  
    const {  
      collection,  
      tokenId,  
      price,  
      expirationTime,  
      marketplace,  
      currency  
    } = params
```

```
    const makerAddress = await signer.getAddress()
```

```
    const listData = await this.reservoir.listToken({  
      maker: makerAddress,  
      source: marketplace || 'opensea.io',  
      params: [{  
        token: `${collection}:${tokenId}`,  
        weiPrice: ethers.parseEther(price.toString()).toString(),  
        expirationTime: Math.floor(expirationTime / 1000).toString(),  
        currency: currency || ethers.ZeroAddress  
      }]  
    })
```

```
    if (listData.steps) {  
      for (const step of listData.steps) {  
        if (step.id === 'nft-approval') {  
          for (const item of step.items) {  
            if (item.status === 'incomplete') {  
              const tx = await signer.sendTransaction({  
                to: item.data.to,  
                data: item.data.data  
              })  
              await tx.wait()
```

```
}  
}  
}
```

```
if (step.id === 'order-signature') {  
  for (const item of step.items) {  
    if (item.status === 'incomplete') {  
      const signature = await signer.signTypedData(  
        item.data.domain,  
        item.data.types,  
        item.data.value  
      )  
      item.data.signature = signature  
    }  
  }  
}  
}  
}
```

```
return { success: true, orderId: listData.orderId }  
}
```

```
async batchList(signer, listings) {  
  const params = listings.map(l => ({  
    token: `${l.collection}:${l.tokenId}`,  
    weiPrice: ethers.parseEther(l.price.toString()).toString(),  
    expirationTime: Math.floor(l.expirationTime / 1000).toString()  
  })))
```

```
const makerAddress = await signer.getAddress()
```

```
const listData = await this.reservoir.listToken({  
  maker: makerAddress,  
  source: 'opensea.io',  
  params: params  
})
```

```
return listData  
}
```



```

async updateListing(signer, collection, tokenId, newPrice) {
return this.listNFT(signer, {
collection,
tokenId,
price: newPrice,
expirationTime: Date.now() + 7 * 24 * 60 * 60 * 1000
})
}
}
}

```

SECTION 8: ROYALTY HANDLING

Royalties have become contentious. Understanding the landscape:

On-Chain Royalties (EIP-2981)

```

contract RoyaltyEnforcingMarketplace {
function executeSale(
address nftContract,
uint256 tokenId,
address seller,
address buyer,
uint256 salePrice
) external payable {
require(msg.value >= salePrice);

(address royaltyReceiver, uint256 royaltyAmount) =
getRoyaltyInfo(
nftContract,
tokenId,
salePrice
);

uint256 sellerAmount = salePrice - royaltyAmount;

if (royaltyAmount > 0 && royaltyReceiver != address(0)) {
payable(royaltyReceiver).transfer(royaltyAmount);
}
}
}

```

```
payable(seller).transfer(sellerAmount);
```

```
IERC721(nftContract).safeTransferFrom(seller, buyer, tokenId);  
}
```

```
function getRoyaltyInfo(  
address nftContract,  
uint256 tokenId,  
uint256 salePrice  
) public view returns (address, uint256) {  
try IERC2981(nftContract).royaltyInfo(tokenId, salePrice) returns (  
address receiver,  
uint256 amount  
) {  
return (receiver, amount);  
} catch {  
return (address(0), 0);  
}  
}  
}
```

```
interface IERC2981 {  
function royaltyInfo(  
uint256 tokenId,  
uint256 salePrice  
) external view returns (address receiver, uint256 royaltyAmount);  
}
```

Operator Filter Registry

OpenSea's approach to enforcing royalties:

```
interface IOperatorFilterRegistry {  
function isOperatorAllowed(address registrant, address operator)  
external view returns (bool);  
function register(address registrant) external;  
function registerAndSubscribe(address registrant, address  
subscription) external;  
}
```

```

contract RoyaltyEnforcingNFT {
    address constant OPERATOR_FILTER_REGISTRY =
0x0000000000000AAeB6D7670E522A718067333cd4E;
    address constant OPENSEA_SUBSCRIPTION =
0x3cc6CddA760b79bAfa08dF41ECFA224f810dCeB6;

    mapping(uint256 => address) private _owners;
    mapping(address => mapping(address => bool)) private
_operatorApprovals;

    bool public operatorFilterEnabled;

    constructor() {
        operatorFilterEnabled = true;

        try
IOperatorFilterRegistry(OPERATOR_FILTER_REGISTRY).registerAndSubscri
address(this),
OPENSEA_SUBSCRIPTION
) {} catch {}
    }

    modifier onlyAllowedOperator(address from) {
        if (operatorFilterEnabled && from != msg.sender) {
            if (!
IOperatorFilterRegistry(OPERATOR_FILTER_REGISTRY).isOperatorAllowed
address(this),
msg.sender
)) {
                revert("Operator not allowed");
            }
        }
        _;
    }

    function setApprovalForAll(address operator, bool approved)
public {
        if (operatorFilterEnabled) {
            require(
IOperatorFilterRegistry(OPERATOR_FILTER_REGISTRY).isOperatorAllowed

```

```

address(this),
operator
),
"Operator not allowed"
);
}
_operatorApprovals[msg.sender][operator] = approved;
}

```

```

function transferFrom(
address from,
address to,
uint256 tokenId
) public onlyAllowedOperator(from) {
require(_isApprovedOrOwner(msg.sender, tokenId));
_transfer(from, to, tokenId);
}

```

```

function _isApprovedOrOwner(address spender, uint256 tokenId)
internal view returns (bool) {
address owner = _owners[tokenId];
return (spender == owner || _operatorApprovals[owner]
[spender]);
}

```

```

function _transfer(address from, address to, uint256 tokenId)
internal {
_owners[tokenId] = to;
}
}

```

SECTION 9: BUILDING YOUR OWN MARKETPLACE

For custom requirements, build marketplace functionality:

```

contract SimpleNFTMarketplace {
struct Listing {
address seller;
address nftContract;

```

```
uint256 tokenId;  
uint256 price;  
address paymentToken;  
uint256 expiresAt;  
bool active;  
}
```

```
struct Offer {  
    address buyer;  
    address nftContract;  
    uint256 tokenId;  
    uint256 amount;  
    address paymentToken;  
    uint256 expiresAt;  
    bool active;  
}
```

```
mapping(bytes32 => Listing) public listings;  
mapping(bytes32 => Offer) public offers;
```

```
uint256 public platformFee;  
address public feeRecipient;  
address public owner;
```

```
event Listed(  
    bytes32 indexed listingId,  
    address indexed seller,  
    address indexed nftContract,  
    uint256 tokenId,  
    uint256 price  
);
```

```
event Sale(  
    bytes32 indexed listingId,  
    address indexed buyer,  
    address indexed seller,  
    uint256 price  
);
```

```
event ListingCancelled(bytes32 indexed listingId);
```

```
event OfferMade(  
bytes32 indexed offerId,  
address indexed buyer,  
address indexed nftContract,  
uint256 tokenId,  
uint256 amount  
);
```

```
event OfferAccepted(bytes32 indexed offerId);  
event OfferCancelled(bytes32 indexed offerId);
```

```
constructor(uint256 fee, address recipient) {  
platformFee = fee;  
feeRecipient = recipient;  
owner = msg.sender;  
}
```

```
function createListing(  
address nftContract,  
uint256 tokenId,  
uint256 price,  
address paymentToken,  
uint256 duration  
) external returns (bytes32) {  
require(price > 0);  
require(IERC721(nftContract).ownerOf(tokenId) == msg.sender);  
require(  
IERC721(nftContract).isApprovedForAll(msg.sender, address(this))  
||  
IERC721(nftContract).getApproved(tokenId) == address(this)  
);
```

```
bytes32 listingId = keccak256(abi.encodePacked(  
msg.sender,  
nftContract,  
tokenId,  
block.timestamp  
));
```

```
listings[listingId] = Listing({
seller: msg.sender,
nftContract: nftContract,
tokenId: tokenId,
price: price,
paymentToken: paymentToken,
expiresAt: block.timestamp + duration,
active: true
});
```

```
emit Listed(listingId, msg.sender, nftContract, tokenId, price);
return listingId;
}
```

```
function buyListing(bytes32 listingId) external payable {
Listing storage listing = listings[listingId];
```

```
require(listing.active);
require(block.timestamp < listing.expiresAt);
require(IERC721(listing.nftContract).ownerOf(listing.tokenId) ==
listing.seller);
```

```
listing.active = false;
```

```
uint256 fee = (listing.price * platformFee) / 10000;
uint256 sellerAmount = listing.price - fee;
```

```
(address royaltyReceiver, uint256 royaltyAmount) =
getRoyaltyInfo(
listing.nftContract,
listing.tokenId,
listing.price
);
```

```
if (royaltyAmount > 0 && royaltyReceiver != address(0)) {
sellerAmount -= royaltyAmount;
}
```

```
if (listing.paymentToken == address(0)) {
require(msg.value >= listing.price);
```

```

if (fee > 0) payable(feeRecipient).transfer(fee);
if (royaltyAmount > 0)
payable(royaltyReceiver).transfer(royaltyAmount);
payable(listing.seller).transfer(sellerAmount);

if (msg.value > listing.price) {
payable(msg.sender).transfer(msg.value - listing.price);
}
} else {
IERC20(listing.paymentToken).transferFrom(msg.sender,
feeRecipient, fee);
if (royaltyAmount > 0) {
IERC20(listing.paymentToken).transferFrom(msg.sender,
royaltyReceiver, royaltyAmount);
}
IERC20(listing.paymentToken).transferFrom(msg.sender,
listing.seller, sellerAmount);
}

IERC721(listing.nftContract).safeTransferFrom(listing.seller,
msg.sender, listing.tokenId);

emit Sale(listingId, msg.sender, listing.seller, listing.price);
}

function cancelListing(bytes32 listingId) external {
Listing storage listing = listings[listingId];
require(listing.seller == msg.sender);
require(listing.active);

listing.active = false;
emit ListingCancelled(listingId);
}

function makeOffer(
address nftContract,
uint256 tokenId,
uint256 amount,
address paymentToken,

```



```
uint256 duration
) external returns (bytes32) {
require(amount > 0);
```

```
bytes32 offerId = keccak256(abi.encodePacked(
msg.sender,
nftContract,
tokenId,
block.timestamp
));
```

```
offers[offerId] = Offer({
buyer: msg.sender,
nftContract: nftContract,
tokenId: tokenId,
amount: amount,
paymentToken: paymentToken,
expiresAt: block.timestamp + duration,
active: true
});
```

```
emit OfferMade(offerId, msg.sender, nftContract, tokenId, amount);
return offerId;
}
```

```
function acceptOffer(bytes32 offerId) external {
Offer storage offer = offers[offerId];
```

```
require(offer.active);
require(block.timestamp < offer.expiresAt);
require(IERC721(offer.nftContract).ownerOf(offer.tokenId) ==
msg.sender);
```

```
offer.active = false;
```

```
uint256 fee = (offer.amount * platformFee) / 10000;
uint256 sellerAmount = offer.amount - fee;
```

```
(address royaltyReceiver, uint256 royaltyAmount) =
getRoyaltyInfo(
```

```
offer.nftContract,  
offer.tokenId,  
offer.amount  
);
```

```
if (royaltyAmount > 0 && royaltyReceiver != address(0)) {  
sellerAmount -= royaltyAmount;  
}
```

```
IERC20(offer.paymentToken).transferFrom(offer.buyer,  
feeRecipient, fee);  
if (royaltyAmount > 0) {  
IERC20(offer.paymentToken).transferFrom(offer.buyer,  
royaltyReceiver, royaltyAmount);  
}  
IERC20(offer.paymentToken).transferFrom(offer.buyer,  
msg.sender, sellerAmount);
```

```
IERC721(offer.nftContract).safeTransferFrom(msg.sender,  
offer.buyer, offer.tokenId);
```

```
emit OfferAccepted(offerId);  
}
```

```
function cancelOffer(bytes32 offerId) external {  
Offer storage offer = offers[offerId];  
require(offer.buyer == msg.sender);  
require(offer.active);
```

```
offer.active = false;  
emit OfferCancelled(offerId);  
}
```

```
function getRoyaltyInfo(  
address nftContract,  
uint256 tokenId,  
uint256 salePrice  
) internal view returns (address, uint256) {  
try IERC2981(nftContract).royaltyInfo(tokenId, salePrice) returns (  
address receiver,
```

```

uint256 amount
) {
if (amount > salePrice / 4) {
return (address(0), 0);
}
return (receiver, amount);
} catch {
return (address(0), 0);
}
}

```

```

function setFee(uint256 newFee) external {
require(msg.sender == owner);
require(newFee <= 500);
platformFee = newFee;
}
}

```

```

interface IERC721 {
function ownerOf(uint256 tokenId) external view returns (address);
function safeTransferFrom(address from, address to, uint256
tokenId) external;
function isApprovedForAll(address owner, address operator)
external view returns (bool);
function getApproved(uint256 tokenId) external view returns
(address);
}

```

```

interface IERC20 {
function transferFrom(address from, address to, uint256 amount)
external returns (bool);
}

```

```

interface IERC2981 {
function royaltyInfo(uint256 tokenId, uint256 salePrice) external
view returns (address, uint256);
}

```

SECTION 10: MARKETPLACE ANALYTICS

Track marketplace activity for insights:

```
class MarketplaceAnalytics {
  constructor(reservoirApi) {
    this.reservoir = reservoirApi
  }

  async getCollectionMetrics(collection) {
    const [tokens, activity, bids] = await Promise.all([
      this.reservoir.getTokens({ collection, limit: 100 }),
      this.reservoir.getCollectionActivity(collection, 100),
      this.reservoir.getBids({ collection, limit: 50 })
    ])

    const floorPrice =
      tokens.tokens[0]?.market?.floorAsk?.price?.amount?.native || 0
    const listedCount = tokens.tokens.filter(t =>
      t.market?.floorAsk).length

    const sales = activity.activities.filter(a => a.type === 'sale')
    const last24hSales = sales.filter(s =>
      Date.now() - new Date(s.timestamp).getTime() < 24 * 60 * 60 *
      1000
    )

    const volume24h = last24hSales.reduce((sum, s) =>
      sum + (s.price?.amount?.native || 0), 0
    )

    const topBid = bids.orders[0]?.price?.amount?.native || 0

    return {
      floorPrice,
      topBid,
      spread: floorPrice - topBid,
      spreadPercent: floorPrice > 0 ? ((floorPrice - topBid) / floorPrice *
      100).toFixed(2) : 0,
      listedCount,
      listingPercent: (listedCount / 100 * 100).toFixed(2),
    }
  }
}
```

```
volume24h,  
sales24h: last24hSales.length  
}  
}
```

```
async getSalesHistory(collection, days = 30) {  
  const sales = await this.reservoir.getSales({  
    collection,  
    limit: 1000  
  })
```

```
  const cutoff = Date.now() - days * 24 * 60 * 60 * 1000
```

```
  const recentSales = sales.sales.filter(s =>  
    new Date(s.timestamp).getTime() > cutoff  
  )
```

```
  const dailyVolumes = {}  
  for (const sale of recentSales) {  
    const date = sale.timestamp.split("T")[0]  
    dailyVolumes[date] = (dailyVolumes[date] || 0) +  
      (sale.price?.amount?.native || 0)  
  }
```

```
  return {  
    totalSales: recentSales.length,  
    totalVolume: recentSales.reduce((sum, s) => sum +  
      (s.price?.amount?.native || 0), 0),  
    dailyVolumes,  
    averagePrice: recentSales.length > 0  
      ? recentSales.reduce((sum, s) => sum + (s.price?.amount?.native  
        || 0), 0) / recentSales.length  
      : 0  
  }  
}
```

```
async findArbitrageOpportunities(collection) {  
  const tokens = await this.reservoir.getTokens({  
    collection,  
    sortBy: 'floorAskPrice',
```

```
limit: 50
```

```
})
```

```
const bids = await this.reservoir.getBids({
```

```
collection,
```

```
sortBy: 'price',
```

```
limit: 50
```

```
})
```

```
const opportunities = []
```

```
for (const token of tokens.tokens) {
```

```
const askPrice = token.market?.floorAsk?.price?.amount?.native
```

```
if (!askPrice) continue
```

```
for (const bid of bids.orders) {
```

```
const bidPrice = bid.price?.amount?.native
```

```
if (bidPrice > askPrice) {
```

```
opportunities.push({
```

```
tokenId: token.token.tokenId,
```

```
askPrice,
```

```
bidPrice,
```

```
profit: bidPrice - askPrice,
```

```
profitPercent: ((bidPrice - askPrice) / askPrice * 100).toFixed(2)
```

```
})
```

```
}
```

```
}
```

```
}
```

```
return opportunities.sort((a, b) => b.profit - a.profit)
```

```
}
```

```
}
```

Marketplace integration enables building sophisticated NFT applications. Whether listing on OpenSea, sweeping floors on Blur, or building custom trading functionality, understanding these protocols gives you the tools to participate in NFT markets programmatically.

CHAPTER 6: DECENTRALIZED STORAGE DEEP DIVE

NFT metadata and assets need homes. Centralized servers fail, disappear, and censor. Decentralized storage solves this by distributing files across networks where no single entity controls access. Your NFT images can survive company bankruptcies, government takedowns, and server failures.

This chapter explores the major decentralized storage systems in depth. IPFS for content-addressed distribution. Arweave for permanent storage. Filecoin for incentivized long-term deals. Understanding these systems helps you choose the right solution for different requirements.

SECTION 1: IPFS ARCHITECTURE

IPFS (InterPlanetary File System) reimagines how files are addressed and distributed. Instead of asking "where is this file?" you ask "what is this file?"

Content Addressing

Traditional web: URL points to location.

`https://server.com/images/cat.png` means "get cat.png from server.com"

IPFS: CID points to content.

`ipfs://QmXnYz...` means "get the file with this hash"

The hash is computed from the file content. Same file always produces same hash. Different file produces different hash. This has profound implications:

Deduplication: Upload the same image twice, get the same CID. The network doesn't store duplicates.

Integrity: Download a file, compute its hash, compare to CID. If they match, you have the correct file. No MITM attacks possible.

Immutability: Change the file, change the hash. Old CID still refers to old content. Versions are automatically preserved.

How IPFS Networks Work

IPFS is peer-to-peer. Every node can serve content it has. When you want a file:

1. Your node broadcasts "who has QmXnYz...?"
2. Nodes with that CID respond
3. You download from nearby nodes
4. Your node caches the content
5. You can now serve it to others

This creates a mesh network where popular content distributes widely. More requests mean more caching mean faster delivery.

The DHT (Distributed Hash Table)

IPFS uses a DHT called Kademlia to track which nodes have which content. Each node maintains a routing table of nearby nodes. Queries propagate through the network until finding nodes with the content.

Key properties:

- Logarithmic lookup time ($\log n$ hops for n nodes)
- Self-organizing and self-healing
- No central authority needed

Pinning and Garbage Collection

By default, IPFS nodes only cache content temporarily. The garbage collector removes old cached content to free space. If nobody pins content, it eventually disappears from the network.

Pinning tells your node "keep this forever." Pinned content survives garbage collection. For NFTs, you must ensure content is pinned somewhere reliable.

SECTION 2: RUNNING YOUR OWN IPFS NODE

For production applications, running your own node provides reliability and control:

Installing IPFS

```
wget https://dist.ipfs.tech/kubo/v0.24.0/kubo_v0.24.0_linux-amd64.tar.gz
```

Breaking down:

wget - Download file from URL
https://dist.ipfs.tech/kubo/v0.24.0/kubo_v0.24.0_linux-amd64.tar.gz - IPFS distribution URL
kubo - Official IPFS implementation name
v0.24.0 - Version number
linux-amd64 - Platform and architecture

```
tar -xzf kubo_v0.24.0_linux-amd64.tar.gz
```

Breaking down:

tar - Archive utility
-x - Extract mode
-z - Decompress gzip
-f - Specify file
kubo_v0.24.0_linux-amd64.tar.gz - Archive file

```
cd kubo && sudo ./install.sh
```

Breaking down:

cd kubo - Change to extracted directory
&& - Run next command if previous succeeds
sudo ./install.sh - Run installer with root privileges

```
ipfs init
```

Breaking down:

ipfs init - Initialize IPFS repository in ~/.ipfs

ipfs daemon

Breaking down:

ipfs daemon - Start IPFS node as background process

Node Configuration

```
import subprocess
import json
```

```
class IPFSNode:
    def __init__(self, api_url = 'http://127.0.0.1:5001'):
        self.api_url = api_url

    def run_command(self, *args):
        result = subprocess.run(['ipfs'] + list(args), capture_output=True,
                                text=True)
        return result.stdout.strip()

    def add_file(self, filepath):
        result = self.run_command('add', '-q', filepath)
        return result

    def add_directory(self, dirpath):
        result = self.run_command('add', '-r', '-q', dirpath)
        lines = result.stdout.strip().split('\n')
        return lines[-1]

    def cat(self, cid):
        return self.run_command('cat', cid)

    def pin_add(self, cid):
        return self.run_command('pin', 'add', cid)

    def pin_rm(self, cid):
        return self.run_command('pin', 'rm', cid)

    def pin_ls(self):
```

```

result = self.run_command('pin', 'ls', '--type = recursive')
pins = []
for line in result.split('\n'):
    if line:
        cid = line.split()[0]
        pins.append(cid)
return pins

```

```

def get_id(self):
    result = self.run_command('id')
    return json.loads(result)

```

```

def get_stats(self):
    repo_stat = self.run_command('repo', 'stat')
    return repo_stat

```

```

def gc(self):
    return self.run_command('repo', 'gc')

```

SECTION 3: IPFS HTTP API

Interact with IPFS nodes via HTTP:

```
import requests
```

```

class IPFSHTTPClient:
    def __init__(self, api_url = 'http://127.0.0.1:5001/api/v0'):
        self.api_url = api_url

    def add_file(self, filepath):
        with open(filepath, 'rb') as f:
            files = {'file': f}
            response = requests.post(f'{self.api_url}/add', files = files)
            return response.json()

    def add_bytes(self, data, filename = 'file'):
        files = {'file': (filename, data)}
        response = requests.post(f'{self.api_url}/add', files = files)
        return response.json()

```

```
def add_json(self, data):
    json_bytes = json.dumps(data).encode()
    return self.add_bytes(json_bytes, 'metadata.json')
```

```
def add_directory(self, dirpath):
    files = []
    for root, dirs, filenames in os.walk(dirpath):
        for filename in filenames:
            filepath = os.path.join(root, filename)
            relative_path = os.path.relpath(filepath, dirpath)
```

```
        with open(filepath, 'rb') as f:
            files.append(('file', (relative_path, f.read())))
```

```
    response = requests.post(
        f'{self.api_url}/add',
        files=files,
        params={'wrap-with-directory': 'true'}
    )
```

```
    results = [json.loads(line) for line in
response.text.strip().split('\n')]
    return results[-1]['Hash']
```

```
def cat(self, cid):
    response = requests.post(f'{self.api_url}/cat', params={'arg': cid})
    return response.content
```

```
def pin_add(self, cid):
    response = requests.post(f'{self.api_url}/pin/add', params={'arg':
cid})
    return response.json()
```

```
def pin_rm(self, cid):
    response = requests.post(f'{self.api_url}/pin/rm', params={'arg':
cid})
    return response.json()
```

```
def pin_ls(self):
```

```
response = requests.post(f'{self.api_url}/pin/ls')
return response.json()
```

```
def resolve(self, path):
    response = requests.post(f'{self.api_url}/resolve', params = {'arg':
path})
    return response.json()
```

```
def name_publish(self, cid):
    response = requests.post(
f'{self.api_url}/name/publish',
params = {'arg': cid}
)
    return response.json()
```

```
def name_resolve(self, peer_id):
    response = requests.post(
f'{self.api_url}/name/resolve',
params = {'arg': peer_id}
)
    return response.json()
```

SECTION 4: ARWEAVE PERMANENT STORAGE

Arweave takes a different approach: pay once, store forever. Instead of ongoing pinning costs, you pay upfront for 200+ years of guaranteed storage.

How Arweave Works

Arweave is a blockchain where miners store data. The storage endowment model calculates how much AR tokens are needed to pay miners for centuries of storage. As storage costs decrease over time (Moore's law), the endowment grows relatively.

Key features:

Permanent by design: No renewal fees. Upload once, stored forever.

Blockweave structure: Each block links to both the previous block and a random historical block, incentivizing miners to store all data.

Wildfire protocol: Nodes that serve data faster earn more rewards, incentivizing good performance.

Proof of Access: Mining requires accessing random historical data, proving miners store the full archive.

Uploading to Arweave

```
import requests
import json
import hashlib
from Crypto.PublicKey import RSA
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256

class ArweaveClient:
    def __init__(self, wallet_path, gateway='https://arweave.net'):
        self.gateway = gateway
        self.wallet = self.load_wallet(wallet_path)

    def load_wallet(self, path):
        with open(path, 'r') as f:
            return json.load(f)

    def get_balance(self):
        address = self.get_address()
        response = requests.get(f'{self.gateway}/wallet/{address}/balance')
        return int(response.text) / 1e12

    def get_address(self):
        n = self.wallet['n']
        n_bytes = self.base64url_decode(n)
        sha256 = hashlib.sha256(n_bytes).digest()
        return self.base64url_encode(sha256)
```

```
def base64url_encode(self, data):
    import base64
    return base64.urlsafe_b64encode(data).rstrip(b'=').decode()
```

```
def base64url_decode(self, data):
    import base64
    padding = 4 - len(data) % 4
    if padding != 4:
        data += '=' * padding
    return base64.urlsafe_b64decode(data)
```

```
def get_price(self, data_size):
    response = requests.get(f'{self.gateway}/price/{data_size}')
    return int(response.text) / 1e12
```

```
def create_transaction(self, data, tags=None):
    last_tx = requests.get(f'{self.gateway}/tx_anchor').text
```

```
    data_encoded = self.base64url_encode(data if isinstance(data,
bytes) else data.encode())
```

```
    tx = {
        'format': 2,
        'last_tx': last_tx,
        'owner': self.wallet['n'],
        'tags': [],
        'target': "",
        'quantity': '0',
        'data': data_encoded,
        'data_size': str(len(data)),
        'data_root': "",
        'reward': ""
    }
```

```
    if tags:
        tx['tags'] = [
            {
                'name': self.base64url_encode(k.encode()),
                'value': self.base64url_encode(v.encode())
            }
        ]
```

```

for k, v in tags.items()
]

return tx

def upload_file(self, filepath, content_type='application/octet-
stream'):
    with open(filepath, 'rb') as f:
        data = f.read()

    tags = {'Content-Type': content_type}
    tx = self.create_transaction(data, tags)

    response = requests.post(
        f'{self.gateway}/tx',
        json=tx,
        headers={'Content-Type': 'application/json'}
    )

    if response.status_code == 200:
        return tx.get('id')
    return None

def get_transaction(self, tx_id):
    response = requests.get(f'{self.gateway}/tx/{tx_id}')
    return response.json()

def get_data(self, tx_id):
    response = requests.get(f'{self.gateway}/{tx_id}')
    return response.content

```

Using Bundlr for Easier Uploads

Bundlr makes Arweave uploads simpler with instant finality:

```

class BundlrClient:
    def __init__(self, api_key, node='https://node1.bundlr.network'):
        self.node = node
        self.api_key = api_key

```



```

def get_headers(self):
    return {
        'Authorization': f'Bearer {self.api_key}',
        'Content-Type': 'application/octet-stream'
    }

def get_price(self, size):
    response = requests.get(f'{self.node}/price/arweave/{size}')
    return response.json()

def upload(self, data, tags = None):
    headers = self.get_headers()

    if tags:
        for key, value in tags.items():
            headers[f'x-tag-{key}'] = value

    response = requests.post(
        f'{self.node}/tx/arweave',
        data = data,
        headers = headers
    )

    return response.json()

def upload_file(self, filepath, content_type):
    with open(filepath, 'rb') as f:
        data = f.read()

    return self.upload(data, {'Content-Type': content_type})

def get_balance(self, address):
    response = requests.get(f'{self.node}/account/balance/arweave/{address}')
    return response.json()

```

SECTION 5: FILECOIN STORAGE DEALS

Filecoin creates a marketplace for storage. Storage providers bid to

store your data, and you pay them over time.

Filecoin Concepts

Storage Providers: Entities that provide disk space. They stake collateral and earn FIL for storing data.

Storage Deals: Agreements between clients and providers. Specify duration, price, and data.

Proof of Replication: Proves the provider stored unique copies of your data.

Proof of Spacetime: Proves the provider continues storing data over time.

Retrieval Deals: Pay providers to retrieve stored data.

Using web3.storage

Web3.storage provides easy Filecoin + IPFS storage:

```
class Web3StorageClient:
    def __init__(self, api_token):
        self.api_token = api_token
        self.base_url = 'https://api.web3.storage'

    def get_headers(self):
        return {
            'Authorization': f'Bearer {self.api_token}'
        }

    def upload_file(self, filepath):
        with open(filepath, 'rb') as f:
            files = {'file': f}
            response = requests.post(
                f'{self.base_url}/upload',
                files=files,
                headers=self.get_headers()
            )
```

```
return response.json()
```

```
def upload_car(self, car_filepath):  
    with open(car_filepath, 'rb') as f:  
        response = requests.post(  
            f'{self.base_url}/car',  
            data=f,  
            headers={  
                **self.get_headers(),  
                'Content-Type': 'application/vnd.ipld.car'  
            }  
        )  
    return response.json()
```

```
def get_status(self, cid):  
    response = requests.get(  
        f'{self.base_url}/status/{cid}',  
        headers=self.get_headers()  
    )  
    return response.json()
```

```
def list_uploads(self):  
    response = requests.get(  
        f'{self.base_url}/user/uploads',  
        headers=self.get_headers()  
    )  
    return response.json()
```

```
def get_deal_info(self, cid):  
    status = self.get_status(cid)  
    return {  
        'cid': cid,  
        'dagSize': status.get('dagSize'),  
        'pins': status.get('pins', []),  
        'deals': status.get('deals', [])  
    }
```

SECTION 6: COMPARING STORAGE SOLUTIONS

Each system has different tradeoffs:

IPFS

Pros:

- Fast retrieval through distributed caching
- Content addressing prevents tampering
- Large ecosystem of gateways and tools
- Free to upload (if you run your own node)

Cons:

- Content disappears without pinning
- Ongoing costs for pinning services
- No native incentive layer

Best for:

- Hot data that needs fast access
- Content that will remain popular
- Applications where you control pinning

Arweave

Pros:

- True permanence (pay once, store forever)
- No ongoing costs
- Immutable by design
- Growing ecosystem

Cons:

- Higher upfront cost
- Slower uploads than IPFS
- Smaller network than IPFS

Best for:

- Important NFT assets that must never disappear
- Historical data and archives
- Legal documents and proofs

Filecoin

Pros:

- Economic incentives ensure storage
- Large storage capacity
- Verifiable storage proofs
- Cheaper for large datasets

Cons:

- Complex deal-making process
- Retrieval can be slow
- Requires understanding of deals

Best for:

- Large datasets (videos, archives)
- Long-term cold storage
- Applications needing verified storage

Comparison Matrix

Feature	IPFS	Arweave	Filecoin
Permanence	Requires pinning	Native	Deal-based
Cost Model	Ongoing	One-time	Deal-based
Retrieval Speed	Fast	Medium	Slow
Capacity	Unlimited	Growing	Massive
Complexity	Low	Medium	High
Best Use	Hot data	Permanent	Large files

SECTION 7: REDUNDANCY STRATEGIES

Don't rely on single storage solution:

```
class RedundantStorage:
```

```
    def __init__(self, pinata_client, arweave_client, web3storage_client):
```

```
        self.pinata = pinata_client
```

```
        self.arweave = arweave_client
```

```
        self.web3storage = web3storage_client
```

```
    def upload_with_redundancy(self, filepath, importance='high'):
```

```
results = {  
'ipfs': None,  
'arweave': None,  
'filecoin': None  
}
```

```
try:  
    ipfs_result = self.pinata.pin_file(filepath)  
    results['ipfs'] = {  
        'cid': ipfs_result['IpfsHash'],  
        'url': f'ipfs://{ipfs_result['IpfsHash']}'  
    }  
except Exception as e:  
    results['ipfs'] = {'error': str(e)}
```

```
if importance in ['high', 'critical']:  
    try:  
        ar_result = self.arweave.upload_file(filepath)  
        results['arweave'] = {  
            'tx_id': ar_result,  
            'url': f'ar://{ar_result}'  
        }  
    except Exception as e:  
        results['arweave'] = {'error': str(e)}
```

```
if importance == 'critical':  
    try:  
        w3s_result = self.web3storage.upload_file(filepath)  
        results['filecoin'] = {  
            'cid': w3s_result.get('cid'),  
            'url': f'ipfs://{w3s_result.get('cid')}'  
        }  
    except Exception as e:  
        results['filecoin'] = {'error': str(e)}
```

```
return results
```

```
def verify_availability(self, storage_results):  
    status = {}
```

```
if storage_results.get('ipfs', {}).get('cid'):
    cid = storage_results['ipfs']['cid']
    status['ipfs'] = self.check_ipfs_availability(cid)
```

```
if storage_results.get('arweave', {}).get('tx_id'):
    tx_id = storage_results['arweave']['tx_id']
    status['arweave'] = self.check_arweave_availability(tx_id)
```

```
if storage_results.get('filecoin', {}).get('cid'):
    cid = storage_results['filecoin']['cid']
    status['filecoin'] = self.check_filecoin_status(cid)
```

```
return status
```

```
def check_ipfs_availability(self, cid):
    gateways = [
        f'https://ipfs.io/ipfs/{cid}',
        f'https://gateway.pinata.cloud/ipfs/{cid}',
        f'https://cloudflare-ipfs.com/ipfs/{cid}'
    ]
```

```
for gateway in gateways:
    try:
        response = requests.head(gateway, timeout=10)
        if response.status_code == 200:
            return {'available': True, 'gateway': gateway}
    except:
        continue
```

```
return {'available': False}
```

```
def check_arweave_availability(self, tx_id):
    try:
        response = requests.head(f'https://arweave.net/{tx_id}',
                                timeout=10)
        return {'available': response.status_code == 200}
    except:
        return {'available': False}
```

```
def check_filecoin_status(self, cid):
```

```

try:
    status = self.web3storage.get_status(cid)
    return {
        'available': True,
        'pins': len(status.get('pins', [])),
        'deals': len(status.get('deals', []))
    }
except:
    return {'available': False}

```

SECTION 8: NFT-SPECIFIC STORAGE PATTERNS

Optimized patterns for NFT projects:

```

class NFTStorageManager:
    def __init__(self, pinata, arweave):
        self.pinata = pinata
        self.arweave = arweave

    def upload_collection(self, images_dir, metadata_dir,
        use_arweave = True):
        print("Uploading images...")
        images_cid = self.upload_images(images_dir, use_arweave)

        print("Generating metadata with image references...")
        self.update_metadata_references(metadata_dir, images_cid,
            use_arweave)

        print("Uploading metadata...")
        metadata_cid = self.upload_metadata(metadata_dir, use_arweave)

        return {
            'images_cid': images_cid,
            'metadata_cid': metadata_cid,
            'base_uri': f'{"ar" if use_arweave else "ipfs"}://{metadata_cid}/'
        }

    def upload_images(self, images_dir, use_arweave):
        if use_arweave:

```



```
return self.upload_directory_arweave(images_dir)
else:
    result = self.pinata.pin_directory(images_dir)
    return result['IpfsHash']
```

```
def upload_metadata(self, metadata_dir, use_arweave):
    if use_arweave:
        return self.upload_directory_arweave(metadata_dir)
    else:
        result = self.pinata.pin_directory(metadata_dir)
        return result['IpfsHash']
```

```
def upload_directory_arweave(self, directory):
    manifest = {'manifest': 'arweave/paths', 'version': '0.1.0', 'paths':
    {}}
```

```
for filename in os.listdir(directory):
    filepath = os.path.join(directory, filename)
    if os.path.isfile(filepath):
        tx_id = self.arweave.upload_file(filepath)
        manifest['paths'][filename] = {'id': tx_id}
```

```
manifest_json = json.dumps(manifest)
manifest_tx = self.arweave.upload_file_from_bytes(
    manifest_json.encode(),
    'application/x.arweave-manifest+json'
)
```

```
return manifest_tx
```

```
def update_metadata_references(self, metadata_dir, images_cid,
use_arweave):
    protocol = 'ar' if use_arweave else 'ipfs'
```

```
for filename in os.listdir(metadata_dir):
    if not filename.endswith('.json'):
        continue
```

```
filepath = os.path.join(metadata_dir, filename)
```

```
with open(filepath, 'r') as f:
    metadata = json.load(f)
```

```
token_id = filename.replace('.json', '')
metadata['image'] = f'{protocol}:{}/{images_cid}/{token_id}.png'
```

```
with open(filepath, 'w') as f:
    json.dump(metadata, f, indent=2)
```

```
def verify_collection_availability(self, base_uri, token_count):
    results = []
    protocol = 'ar' if base_uri.startswith('ar://') else 'ipfs'
```

```
    for token_id in range(1, min(token_count + 1, 11)):
        metadata_url = self.resolve_uri(f'{base_uri}/{token_id}.json')
```

```
    try:
        response = requests.get(metadata_url, timeout=30)
        if response.status_code == 200:
            metadata = response.json()
            image_url = self.resolve_uri(metadata.get('image', ''))
```

```
        image_response = requests.head(image_url, timeout=30)
        image_ok = image_response.status_code == 200
```

```
        results.append({
            'token_id': token_id,
            'metadata_ok': True,
            'image_ok': image_ok
        })
    else:
        results.append({
            'token_id': token_id,
            'metadata_ok': False,
            'error': f'Status {response.status_code}'
        })
    except Exception as e:
        results.append({
            'token_id': token_id,
            'metadata_ok': False,
```

```
'error': str(e)
})
```

return results

```
def resolve_uri(self, uri):
    if uri.startswith('ipfs://'):
        return uri.replace('ipfs://', 'https://ipfs.io/ipfs/')
    if uri.startswith('ar://'):
        return uri.replace('ar://', 'https://arweave.net/')
    return uri
```

SECTION 9: COST OPTIMIZATION

Storage costs add up for large collections:

```
class StorageCostCalculator:
    def __init__(self):
        self.ipfs_pinning_rate = 0.15
        self.arweave_rate_per_mb = 0.005

    def calculate_ipfs_monthly_cost(self, total_size_mb):
        return total_size_mb * self.ipfs_pinning_rate / 1000

    def calculate_arweave_one_time_cost(self, total_size_mb):
        return total_size_mb * self.arweave_rate_per_mb

    def calculate_collection_costs(self, image_count, avg_image_size_kb,
        avg_metadata_size_kb):
        total_image_size_mb = (image_count * avg_image_size_kb) / 1024
        total_metadata_size_mb = (image_count * avg_metadata_size_kb) /
        1024
        total_size_mb = total_image_size_mb + total_metadata_size_mb

    ipfs_monthly = self.calculate_ipfs_monthly_cost(total_size_mb)
    ipfs_yearly = ipfs_monthly * 12
    ipfs_5year = ipfs_yearly * 5

    arweave_once =
```

```

self.calculate_arweave_one_time_cost(total_size_mb)

return {
    'total_size_mb': total_size_mb,
    'ipfs': {
        'monthly': ipfs_monthly,
        'yearly': ipfs_yearly,
        '5_year': ipfs_5year
    },
    'arweave': {
        'one_time': arweave_once
    },
    'recommendation': 'arweave' if arweave_once < ipfs_5year else 'ipfs'
}

def optimize_image_sizes(self, target_total_cost, image_count):
    target_size_mb = target_total_cost / self.arweave_rate_per_mb
    target_size_kb = (target_size_mb * 1024) / image_count

    return {
        'max_image_size_kb': int(target_size_kb * 0.9),
        'max_metadata_size_kb': int(target_size_kb * 0.1)
    }

def optimize_images_for_storage(images_dir, output_dir,
                                max_size_kb):
    from PIL import Image
    import os

    os.makedirs(output_dir, exist_ok=True)

    for filename in os.listdir(images_dir):
        if not filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            continue

        input_path = os.path.join(images_dir, filename)
        output_path = os.path.join(output_dir, filename)

        img = Image.open(input_path)

```

```

quality = 95
while quality > 10:
    img.save(output_path, quality=quality, optimize=True)
    size_kb = os.path.getsize(output_path) / 1024

    if size_kb <= max_size_kb:
        break

    quality -= 5

if quality <= 10 and size_kb > max_size_kb:
    width, height = img.size
    img = img.resize((int(width * 0.8), int(height * 0.8)),
Image.LANCZOS)
    quality = 85

```

SECTION 10: MONITORING AND MAINTENANCE

Ensure stored content remains available:

```

class StorageMonitor:
    def __init__(self, storage_records):
        self.records = storage_records
        self.alert_callback = None

    def set_alert_callback(self, callback):
        self.alert_callback = callback

    def check_all_content(self):
        results = []

        for record in self.records:
            status = self.check_content(record)
            results.append(status)

        if not status['available'] and self.alert_callback:
            self.alert_callback(f'Content unavailable: {record["cid"]}')

```

return results

```
def check_content(self, record):  
    cid = record['cid']  
    storage_type = record['type']
```

```
    if storage_type == 'ipfs':  
        return self.check_ipfs(cid)  
    elif storage_type == 'arweave':  
        return self.check_arweave(cid)  
    elif storage_type == 'filecoin':  
        return self.check_filecoin(cid)
```

```
    return {'cid': cid, 'available': False, 'error': 'Unknown type'}
```

```
def check_ipfs(self, cid):  
    gateways = [  
        'https://ipfs.io/ipfs/',  
        'https://gateway.pinata.cloud/ipfs/',  
        'https://cloudflare-ipfs.com/ipfs/'  
    ]
```

```
    for gateway in gateways:  
        try:  
            response = requests.head(f'{gateway}{cid}', timeout=15)  
            if response.status_code == 200:  
                return {  
                    'cid': cid,  
                    'type': 'ipfs',  
                    'available': True,  
                    'gateway': gateway  
                }  
        except:  
            continue
```

```
    return {'cid': cid, 'type': 'ipfs', 'available': False}
```

```
def check_arweave(self, tx_id):  
    try:  
        response = requests.head(f'https://arweave.net/{tx_id}',
```

```

timeout = 15)
return {
'cid': tx_id,
'type': 'arweave',
'available': response.status_code == 200
}
except Exception as e:
return {
'cid': tx_id,
'type': 'arweave',
'available': False,
'error': str(e)
}

def check_filecoin(self, cid):
try:
response = requests.get(
f'https://api.web3.storage/status/{cid}',
timeout=15
)
data = response.json()

deals = data.get('deals', [])
active_deals = [d for d in deals if d.get('status') == 'Active']

return {
'cid': cid,
'type': 'filecoin',
'available': len(active_deals) > 0,
'deal_count': len(active_deals)
}
except Exception as e:
return {
'cid': cid,
'type': 'filecoin',
'available': False,
'error': str(e)
}

def generate_health_report(self):

```

```

results = self.check_all_content()

total = len(results)
available = sum(1 for r in results if r.get('available'))
unavailable = total - available

by_type = {}
for result in results:
    t = result.get('type', 'unknown')
    if t not in by_type:
        by_type[t] = {'total': 0, 'available': 0}
    by_type[t]['total'] += 1
    if result.get('available'):
        by_type[t]['available'] += 1

return {
    'total': total,
    'available': available,
    'unavailable': unavailable,
    'availability_percent': (available / total * 100) if total > 0 else 0,
    'by_type': by_type,
    'unavailable_items': [r for r in results if not r.get('available')]
}

```

Decentralized storage ensures your NFT assets survive beyond any single company or server. Choose IPFS for speed and flexibility, Arweave for guaranteed permanence, or Filecoin for large-scale verified storage. For important collections, use multiple systems for redundancy. Monitor regularly to catch issues before users notice. Your digital assets deserve infrastructure as durable as the blockchain itself.

CHAPTER 7: ADVANCED NFT PATTERNS

Basic NFTs are static: mint once, unchanging forever. But the technology enables far more sophisticated patterns. NFTs that evolve over time. Tokens that cannot transfer. Expensive assets split into affordable pieces. NFTs that combine into new creations.

This chapter explores advanced patterns that push NFT capabilities

beyond simple collectibles. Each pattern solves real problems and opens new possibilities for digital ownership.

SECTION 1: DYNAMIC NFTS

Dynamic NFTs change based on external conditions. A game character that levels up. Weather NFTs that reflect real conditions. Sports cards that update with player statistics.

On-Chain Dynamic Metadata

```
contract DynamicNFT {
    struct TokenAttributes {
        uint256 level;
        uint256 experience;
        uint256 strength;
        uint256 agility;
        uint256 lastUpdated;
    }

    mapping(uint256 => TokenAttributes) public attributes;
    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;

    address public gameContract;
    address public owner;
    uint256 private _currentTokenId;

    string[] private levelNames = ["Novice", "Apprentice",
    "Journeyman", "Expert", "Master", "Legendary"];

    event AttributesUpdated(uint256 indexed tokenId, uint256 level,
    uint256 experience);

    constructor() {
        owner = msg.sender;
    }

    function setGameContract(address game) external {
```

```
require(msg.sender == owner);
gameContract = game;
}
```

```
function mint(address to) external returns (uint256) {
    uint256 tokenId = _currentTokenId;
    _currentTokenId ++;
```

```
    _owners[tokenId] = to;
    _balances[to] ++;
```

```
    attributes[tokenId] = TokenAttributes({
        level: 1,
        experience: 0,
        strength: 10,
        agility: 10,
        lastUpdated: block.timestamp
    });
```

```
    return tokenId;
}
```

```
function addExperience(uint256 tokenId, uint256 amount) external
{
    require(msg.sender == gameContract);
    require(_owners[tokenId] != address(0));
```

```
    TokenAttributes storage attr = attributes[tokenId];
    attr.experience += amount;
```

```
    uint256 expForNextLevel = attr.level * 100;
    while (attr.experience >= expForNextLevel && attr.level < 6) {
        attr.level ++;
        attr.strength += 5;
        attr.agility += 3;
        expForNextLevel = attr.level * 100;
    }
```

```
    attr.lastUpdated = block.timestamp;
    emit AttributesUpdated(tokenId, attr.level, attr.experience);
```

```
}
```

```
function tokenURI(uint256 tokenId) public view returns (string  
memory) {  
    require(_owners[tokenId] != address(0));
```

```
    TokenAttributes memory attr = attributes[tokenId];  
    string memory levelName = levelNames[attr.level - 1];
```

```
    string memory json = string(abi.encodePacked(  
        '{"name":"Hero #', toString(tokenId),  
        '", "description":"A dynamic hero NFT that levels up through  
gameplay",',  
        '"attributes":[',  
        '{"trait_type":"Level", "value":', levelName, '}',',',  
        '{"trait_type":"Level Number", "display_type":"number", "value":',  
toString(attr.level), '}',',',  
        '{"trait_type":"Experience", "display_type":"number", "value":',  
toString(attr.experience), '}',',',  
        '{"trait_type":"Strength", "display_type":"number", "value":',  
toString(attr.strength), '}',',',  
        '{"trait_type":"Agility", "display_type":"number", "value":',  
toString(attr.agility), '}',',  
        '], "image":', generateSVG(tokenId, attr), '"}'  
    ));
```

```
    return string(abi.encodePacked(  
        'data:application/json;base64,',  
        encode(bytes(json))  
    ));  
}
```

```
function generateSVG(uint256 tokenId, TokenAttributes memory  
attr) internal pure returns (string memory) {  
    string memory color = attr.level >= 5 ? "#FFD700" :  
    attr.level >= 3 ? "#C0C0C0" : "#CD7F32";
```

```
    string memory svg = string(abi.encodePacked(  
        'data:image/svg+xml;base64',  
        encode(bytes(string(abi.encodePacked(
```

```
'<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 400
400">',
'<rect width="400" height="400" fill="#1a1a2e"/>',
'<circle cx="200" cy="150" r="80" fill="", color,""/>',
'<text x="200" y="280" text-anchor="middle" fill="white" font-
size="24">Level ', toString(attr.level), '</text>',
'<text x="200" y="320" text-anchor="middle" fill="white" font-
size="16">STR: ', toString(attr.strength), ' AGI: ',
toString(attr.agility), '</text>',
'</svg>'
))))
));
```

```
return svg;
}
```

```
function toString(uint256 value) internal pure returns (string
memory) {
    if (value == 0) return "0";
    uint256 temp = value;
    uint256 digits;
    while (temp != 0) { digits++; temp /= 10; }
    bytes memory buffer = new bytes(digits);
    while (value != 0) {
        digits -= 1;
        buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
        value /= 10;
    }
    return string(buffer);
}
```

```
function encode(bytes memory data) internal pure returns (string
memory) {
    string memory TABLE =
"ABCDEFGHGIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345
if (data.length == 0) return "";
uint256 encodedLen = 4 * ((data.length + 2) / 3);
bytes memory result = new bytes(encodedLen);
bytes memory table = bytes(TABLE);
uint256 i = 0;
```

```

uint256 j = 0;
while (i < data.length) {
    uint256 a = uint8(data[i + +]);
    uint256 b = i < data.length ? uint8(data[i + +]) : 0;
    uint256 c = i < data.length ? uint8(data[i + +]) : 0;
    uint256 triple = (a << 16) + (b << 8) + c;
    result[j + +] = table[(triple >> 18) & 0x3F];
    result[j + +] = table[(triple >> 12) & 0x3F];
    result[j + +] = table[(triple >> 6) & 0x3F];
    result[j + +] = table[triple & 0x3F];
}
uint256 mod = data.length % 3;
if (mod > 0) { result[encodedLen - 1] = "="; if (mod == 1)
result[encodedLen - 2] = "="; }
return string(result);
}

```

```

function ownerOf(uint256 tokenId) public view returns (address) {
    return _owners[tokenId];
}

```

```

function balanceOf(address account) public view returns (uint256)
{
    return _balances[account];
}
}

```

Oracle-Based Dynamic NFTs

```

contract WeatherNFT {
    struct WeatherData {
        int256 temperature;
        uint256 humidity;
        string condition;
        uint256 lastUpdated;
    }
}

```

```

mapping(uint256 => string) public tokenLocations;
mapping(uint256 => WeatherData) public tokenWeather;
mapping(uint256 => address) private _owners;

```

```
address public oracle;  
address public owner;  
uint256 private _currentTokenId;
```

```
event WeatherUpdated(uint256 indexed tokenId, string condition,  
int256 temperature);
```

```
constructor(address oracleAddress) {  
    owner = msg.sender;  
    oracle = oracleAddress;  
}
```

```
function mint(address to, string memory location) external returns  
(uint256) {  
    uint256 tokenId = _currentTokenId;  
    _currentTokenId ++;
```

```
    _owners[tokenId] = to;  
    tokenLocations[tokenId] = location;
```

```
    return tokenId;  
}
```

```
function updateWeather(  
    uint256 tokenId,  
    int256 temperature,  
    uint256 humidity,  
    string memory condition  
) external {  
    require(msg.sender == oracle);  
    require(_owners[tokenId] != address(0));
```

```
    tokenWeather[tokenId] = WeatherData({  
        temperature: temperature,  
        humidity: humidity,  
        condition: condition,  
        lastUpdated: block.timestamp  
    });
```

```
emit WeatherUpdated(tokenId, condition, temperature);  
}
```

```
function tokenId(uint256 tokenId) public view returns (string  
memory) {  
    require(_owners[tokenId] != address(0));
```

```
    WeatherData memory weather = tokenWeather[tokenId];  
    string memory location = tokenLocations[tokenId];
```

```
    string memory bgColor = getBackgroundColor(weather.condition);  
    string memory icon = getWeatherIcon(weather.condition);
```

```
    string memory svg = string(abi.encodePacked(  
        '<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 400  
400">',  
        '<rect width="400" height="400" fill="", bgColor, "/>',  
        '<text x="200" y="100" text-anchor="middle" font-size="60">',  
        icon, '</text>',  
        '<text x="200" y="200" text-anchor="middle" fill="white" font-  
size="48">', intToString(weather.temperature), ' C</text>',  
        '<text x="200" y="280" text-anchor="middle" fill="white" font-  
size="24">', weather.condition, '</text>',  
        '<text x="200" y="350" text-anchor="middle" fill="white" font-  
size="18">', location, '</text>',  
        '</svg>'  
    ));
```

```
    return buildTokenURI(tokenId, location, weather, svg);  
}
```

```
function getBackgroundColor(string memory condition) internal  
pure returns (string memory) {  
    bytes32 conditionHash = keccak256(bytes(condition));  
    if (conditionHash == keccak256(bytes("Sunny"))) return  
"#87CEEB";  
    if (conditionHash == keccak256(bytes("Cloudy"))) return  
"#708090";  
    if (conditionHash == keccak256(bytes("Rainy"))) return  
"#4682B4";
```

```

if (conditionHash == keccak256(bytes("Snowy"))) return
"#EOFFFF";
return "#1a1a2e";
}

```

```

function getWeatherIcon(string memory condition) internal pure
returns (string memory) {
    bytes32 conditionHash = keccak256(bytes(condition));
    if (conditionHash == keccak256(bytes("Sunny"))) return
    unicode"☀️";
    if (conditionHash == keccak256(bytes("Cloudy"))) return
    unicode"☁️";
    if (conditionHash == keccak256(bytes("Rainy"))) return
    unicode"🌧️";
    if (conditionHash == keccak256(bytes("Snowy"))) return
    unicode"❄️";
    return unicode"❄️";
}

```

```

function buildTokenURI(
    uint256 tokenId,
    string memory location,
    WeatherData memory weather,
    string memory svg
) internal pure returns (string memory) {
    return "";
}

```

```

function intToString(int256 value) internal pure returns (string
memory) {
    if (value == 0) return "0";
    bool negative = value < 0;
    uint256 absValue = negative ? uint256(-value) : uint256(value);
    string memory str = uintToString(absValue);
    if (negative) {
        return string(abi.encodePacked("-", str));
    }
    return str;
}

```



```

function uintToString(uint256 value) internal pure returns (string
memory) {
    if (value == 0) return "0";
    uint256 temp = value;
    uint256 digits;
    while (temp != 0) { digits ++; temp /= 10; }
    bytes memory buffer = new bytes(digits);
    while (value != 0) {
        digits -= 1;
        buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
        value /= 10;
    }
    return string(buffer);
}

```

```

function ownerOf(uint256 tokenId) public view returns (address) {
    return _owners[tokenId];
}
}

```

SECTION 2: SOULBOUND TOKENS (SBTs)

Soulbound tokens cannot transfer after minting. They represent non-transferable credentials, achievements, or identity attributes.

```

contract SoulboundToken {
    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;
    mapping(uint256 => bool) private _soulbound;

    string public name;
    string public symbol;

    address public issuer;
    uint256 private _currentTokenId;

    event Attest(address indexed to, uint256 indexed tokenId);
    event Revoke(address indexed from, uint256 indexed tokenId);
}

```

```
constructor(string memory tokenName, string memory  
tokenSymbol) {  
    name = tokenName;  
    symbol = tokenSymbol;  
    issuer = msg.sender;  
}
```

```
function attest(address to) external returns (uint256) {  
    require(msg.sender == issuer);  
    require(to != address(0));
```

```
    uint256 tokenId = _currentTokenId;  
    _currentTokenId ++;
```

```
    _owners[tokenId] = to;  
    _balances[to] ++;  
    _soulbound[tokenId] = true;
```

```
    emit Attest(to, tokenId);  
    return tokenId;  
}
```

```
function revoke(uint256 tokenId) external {  
    require(msg.sender == issuer);  
    require(_owners[tokenId] != address(0));
```

```
    address from = _owners[tokenId];  
    _balances[from]--;  
    delete _owners[tokenId];
```

```
    emit Revoke(from, tokenId);  
}
```

```
function burn(uint256 tokenId) external {  
    require(_owners[tokenId] == msg.sender);
```

```
    _balances[msg.sender]--;  
    delete _owners[tokenId];
```

```
    emit Revoke(msg.sender, tokenId);
```

```
}
```

```
function ownerOf(uint256 tokenId) public view returns (address) {  
    address tokenOwner = _owners[tokenId];  
    require(tokenOwner != address(0));  
    return tokenOwner;  
}
```

```
function balanceOf(address account) public view returns (uint256)  
{  
    require(account != address(0));  
    return _balances[account];  
}
```

```
function isSoulbound(uint256 tokenId) public view returns (bool) {  
    return _soulbound[tokenId];  
}
```

```
function transferFrom(address, address, uint256 tokenId) public  
pure {  
    revert("Soulbound: non-transferable");  
}
```

```
function safeTransferFrom(address, address, uint256) public pure {  
    revert("Soulbound: non-transferable");  
}
```

```
function approve(address, uint256) public pure {  
    revert("Soulbound: non-transferable");  
}
```

```
function setApprovalForAll(address, bool) public pure {  
    revert("Soulbound: non-transferable");  
}
```

```
function getApproved(uint256) public pure returns (address) {  
    return address(0);  
}
```

```
function isApprovedForAll(address, address) public pure returns
```

```
(bool) {  
    return false;  
}
```

```
function supportsInterface(bytes4 interfaceId) public pure returns  
(bool) {  
    return interfaceId == 0x01ffc9a7 || interfaceId == 0x80ac58cd;  
}  
}
```

Credential System with SBTs

```
contract CredentialSystem {  
    struct Credential {  
        string credentialType;  
        string issuerName;  
        uint256 issuedAt;  
        uint256 expiresAt;  
        bytes32 dataHash;  
    }
```

```
    mapping(uint256 => Credential) public credentials;  
    mapping(uint256 => address) private _owners;  
    mapping(address => uint256[]) private _ownedCredentials;  
    mapping(address => bool) public authorizedIssuers;
```

```
    address public admin;  
    uint256 private _currentTokenId;
```

```
    event CredentialIssued(  
        address indexed to,  
        uint256 indexed tokenId,  
        string credentialType,  
        string issuerName  
    );
```

```
    event CredentialRevoked(uint256 indexed tokenId, string reason);
```

```
    constructor() {  
        admin = msg.sender;
```

```
authorizedIssuers[msg.sender] = true;  
}
```

```
modifier onlyIssuer() {  
    require(authorizedIssuers[msg.sender]);  
    _;  
}
```

```
function authorizeIssuer(address issuer) external {  
    require(msg.sender == admin);  
    authorizedIssuers[issuer] = true;  
}
```

```
function revokeIssuer(address issuer) external {  
    require(msg.sender == admin);  
    authorizedIssuers[issuer] = false;  
}
```

```
function issueCredential(  
    address to,  
    string memory credentialType,  
    string memory issuerName,  
    uint256 validityPeriod,  
    bytes32 dataHash  
) external onlyIssuer returns (uint256) {  
    uint256 tokenId = _currentTokenId;  
    _currentTokenId ++;
```

```
_owners[tokenId] = to;  
_ownedCredentials[to].push(tokenId);
```

```
credentials[tokenId] = Credential({  
    credentialType: credentialType,  
    issuerName: issuerName,  
    issuedAt: block.timestamp,  
    expiresAt: validityPeriod > 0 ? block.timestamp + validityPeriod :  
    0,  
    dataHash: dataHash  
});
```

```
emit CredentialIssued(to, tokenId, credentialType, issuerName);
return tokenId;
}
```

```
function revokeCredential(uint256 tokenId, string memory reason)
external onlyIssuer {
    require(_owners[tokenId] != address(0));
    delete _owners[tokenId];
    emit CredentialRevoked(tokenId, reason);
}
```

```
function verifyCredential(uint256 tokenId) public view returns (
    bool valid,
    address holder,
    string memory credentialType,
    bool expired
) {
    holder = _owners[tokenId];
    if (holder == address(0)) {
        return (false, address(0), "", false);
    }
}
```

```
Credential memory cred = credentials[tokenId];
expired = cred.expiresAt > 0 && block.timestamp >
cred.expiresAt;
valid = !expired;
credentialType = cred.credentialType;
}
```

```
function getCredentials(address holder) external view returns
(uint256[] memory) {
    return _ownedCredentials[holder];
}
```

```
function hasValidCredential(
    address holder,
    string memory credentialType
) external view returns (bool) {
    uint256[] memory creds = _ownedCredentials[holder];
```

```

for (uint256 i = 0; i < creds.length; i++) {
    if (_owners[creds[i]] == holder) {
        Credential memory c = credentials[creds[i]];
        if (keccak256(bytes(c.credentialType)) ==
            keccak256(bytes(credentialType))) {
            if (c.expiresAt == 0 || block.timestamp <= c.expiresAt) {
                return true;
            }
        }
    }
}

return false;
}

function ownerOf(uint256 tokenId) public view returns (address) {
    return _owners[tokenId];
}
}

```

SECTION 3: NFT FRACTIONALIZATION

Split expensive NFTs into affordable fungible pieces:

```

contract FractionalizedNFT {
    address public nftContract;
    uint256 public nftTokenId;
    address public originalOwner;

    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public
    allowance;

    bool public nftDeposited;

```

```
bool public nftRedeemed;
```

```
uint256 public buyoutPrice;
```

```
address public buyoutProposer;
```

```
uint256 public buyoutDeadline;
```

```
uint256 public constant BUYOUT_DURATION = 7 days;
```

```
event Transfer(address indexed from, address indexed to, uint256 value);
```

```
event Approval(address indexed owner, address indexed spender, uint256 value);
```

```
event NFTDeposited(address indexed depositor, address indexed nftContract, uint256 tokenId);
```

```
event NFTRedeemed(address indexed redeemer);
```

```
event BuyoutProposed(address indexed proposer, uint256 price);
```

```
event BuyoutCompleted(address indexed buyer, uint256 price);
```

```
constructor(
```

```
string memory tokenName,
```

```
string memory tokenSymbol,
```

```
uint256 supply
```

```
) {
```

```
name = tokenName;
```

```
symbol = tokenSymbol;
```

```
decimals = 18;
```

```
totalSupply = supply;
```

```
}
```

```
function depositNFT(address nft, uint256 tokenId) external {  
    require(!nftDeposited);
```

```
IERC721(nft).transferFrom(msg.sender, address(this), tokenId);
```

```
nftContract = nft;
```

```
nftTokenId = tokenId;
```

```
originalOwner = msg.sender;
```

```
nftDeposited = true;
```

```
balanceOf[msg.sender] = totalSupply;
```



```
emit NFTDeposited(msg.sender, nft, tokenId);
emit Transfer(address(0), msg.sender, totalSupply);
}
```

```
function redeem() external {
    require(nftDeposited);
    require(!nftRedeemed);
    require(balanceOf[msg.sender] == totalSupply);
```

```
    nftRedeemed = true;
    balanceOf[msg.sender] = 0;
```

```
    IERC721(nftContract).transferFrom(address(this), msg.sender,
nftTokenId);
```

```
    emit NFTRedeemed(msg.sender);
    emit Transfer(msg.sender, address(0), totalSupply);
}
```

```
function proposeBuyout() external payable {
    require(nftDeposited);
    require(!nftRedeemed);
    require(msg.value > 0);
```

```
    if (buyoutProposer != address(0)) {
        require(msg.value > buyoutPrice);
        payable(buyoutProposer).transfer(buyoutPrice);
    }
```

```
    buyoutPrice = msg.value;
    buyoutProposer = msg.sender;
    buyoutDeadline = block.timestamp + BUYOUT_DURATION;
```

```
    emit BuyoutProposed(msg.sender, msg.value);
}
```

```
function executeBuyout() external {
    require(buyoutProposer == msg.sender);
    require(block.timestamp >= buyoutDeadline);
    require(nftDeposited);
```

```
require(!nftRedeemed);
```

```
nftRedeemed = true;
```

```
IERC721(nftContract).transferFrom(address(this), msg.sender,  
nftTokenId);
```

```
emit BuyoutCompleted(msg.sender, buyoutPrice);  
}
```

```
function claimProceeds() external {  
    require(nftRedeemed);  
    require(buyoutPrice > 0);
```

```
    uint256 userBalance = balanceOf[msg.sender];  
    require(userBalance > 0);
```

```
    uint256 share = (buyoutPrice * userBalance) / totalSupply;  
    balanceOf[msg.sender] = 0;
```

```
    payable(msg.sender).transfer(share);
```

```
    emit Transfer(msg.sender, address(0), userBalance);  
}
```

```
function transfer(address to, uint256 amount) external returns  
(bool) {  
    require(balanceOf[msg.sender] >= amount);
```

```
    balanceOf[msg.sender] -= amount;  
    balanceOf[to] += amount;
```

```
    emit Transfer(msg.sender, to, amount);  
    return true;  
}
```

```
function approve(address spender, uint256 amount) external  
returns (bool) {  
    allowance[msg.sender][spender] = amount;  
    emit Approval(msg.sender, spender, amount);
```

```
return true;
}
```

```
function transferFrom(address from, address to, uint256 amount)
external returns (bool) {
    require(balanceOf[from] >= amount);
    require(allowance[from][msg.sender] >= amount);

    balanceOf[from] -= amount;
    balanceOf[to] += amount;
    allowance[from][msg.sender] -= amount;
```

```
    emit Transfer(from, to, amount);
    return true;
}
}
```

```
interface IERC721 {
    function transferFrom(address from, address to, uint256 tokenId)
external;
    function ownerOf(uint256 tokenId) external view returns (address);
}
```

SECTION 4: NFT LENDING AND COLLATERAL

Use NFTs as collateral for loans:

```
contract NFTLending {
    struct Loan {
        address borrower;
        address nftContract;
        uint256 nftTokenId;
        uint256 loanAmount;
        uint256 interestRate;
        uint256 duration;
        uint256 startTime;
        bool active;
        bool defaulted;
    }
```

```
mapping(uint256 => Loan) public loans;  
uint256 public nextLoanId;
```

```
mapping(address => mapping(uint256 => uint256)) public  
nftToLoan;
```

```
mapping(address => uint256[]) public borrowerLoans;
```

```
uint256 public constant RATE_DENOMINATOR = 10000;
```

```
address public owner;
```

```
event LoanCreated(  
    uint256 indexed loanId,  
    address indexed borrower,  
    address nftContract,  
    uint256 nftTokenId,  
    uint256 amount  
);
```

```
event LoanRepaid(uint256 indexed loanId, uint256 totalRepaid);  
event LoanDefaulted(uint256 indexed loanId);  
event NFTClaimed(uint256 indexed loanId, address indexed  
claimer);
```

```
constructor() {  
    owner = msg.sender;  
}
```

```
function createLoan(  
    address nftContract,  
    uint256 nftTokenId,  
    uint256 loanAmount,  
    uint256 interestRate,  
    uint256 duration  
) external payable returns (uint256) {  
    require(msg.value >= loanAmount);
```

```
    IERC721(nftContract).transferFrom(msg.sender, address(this),  
nftTokenId);
```

```
uint256 loanId = nextLoanId;  
nextLoanId ++;
```

```
loans[loanId] = Loan({  
  borrower: msg.sender,  
  nftContract: nftContract,  
  nftTokenId: nftTokenId,  
  loanAmount: loanAmount,  
  interestRate: interestRate,  
  duration: duration,  
  startTime: block.timestamp,  
  active: true,  
  defaulted: false  
});
```

```
nftToLoan[nftContract][nftTokenId] = loanId;  
borrowerLoans[msg.sender].push(loanId);
```

```
emit LoanCreated(loanId, msg.sender, nftContract, nftTokenId,  
loanAmount);  
return loanId;  
}
```

```
function fundLoan(uint256 loanId) external payable {  
  Loan storage loan = loans[loanId];  
  require(loan.active);  
  require(loan.startTime == 0);  
  require(msg.value >= loan.loanAmount);
```

```
  loan.startTime = block.timestamp;
```

```
  payable(loan.borrower).transfer(loan.loanAmount);  
}
```

```
function repayLoan(uint256 loanId) external payable {  
  Loan storage loan = loans[loanId];  
  require(loan.active);  
  require(loan.borrower == msg.sender);
```

```
uint256 totalOwed = calculateTotalOwed(loanId);
require(msg.value >= totalOwed);
```

```
loan.active = false;
```

```
IERC721(loan.nftContract).transferFrom(address(this), msg.sender,
loan.nftTokenId);
```

```
if (msg.value > totalOwed) {
payable(msg.sender).transfer(msg.value - totalOwed);
}
```

```
emit LoanRepaid(loanId, totalOwed);
}
```

```
function claimDefaultedNFT(uint256 loanId) external {
Loan storage loan = loans[loanId];
require(loan.active);
require(block.timestamp > loan.startTime + loan.duration);
```

```
loan.active = false;
loan.defaulted = true;
```

```
IERC721(loan.nftContract).transferFrom(address(this), msg.sender,
loan.nftTokenId);
```

```
emit LoanDefaulted(loanId);
emit NFTClaimed(loanId, msg.sender);
}
```

```
function calculateTotalOwed(uint256 loanId) public view returns
(uint256) {
Loan memory loan = loans[loanId];
```

```
uint256 timeElapsed = block.timestamp - loan.startTime;
uint256 interest = (loan.loanAmount * loan.interestRate *
timeElapsed) /
(RATE_DENOMINATOR * 365 days);
```

```
return loan.loanAmount + interest;
```

```
}
```

```
function getLoanStatus(uint256 loanId) external view returns (  
    bool active,  
    bool defaulted,  
    uint256 totalOwed,  
    uint256 timeRemaining  
) {
```

```
    Loan memory loan = loans[loanId];  
    active = loan.active;  
    defaulted = loan.defaulted;  
    totalOwed = active ? calculateTotalOwed(loanId) : 0;
```

```
    if (active && loan.startTime + loan.duration > block.timestamp) {  
        timeRemaining = loan.startTime + loan.duration -  
        block.timestamp;  
    }  
    }  
}
```

SECTION 5: COMPOSABLE NFTS

NFTs that combine or contain other NFTs:

```
contract ComposableNFT {  
    struct TokenComposition {  
        uint256[] childTokenIds;  
        address[] childContracts;  
    }  
}
```

```
mapping(uint256 => TokenComposition) private _compositions;  
mapping(uint256 => address) private _owners;  
mapping(address => uint256) private _balances;  
mapping(address => mapping(uint256 => uint256)) private  
_childToParent;
```

```
address public owner;  
uint256 private _currentTokenId;
```

```
event Composed(uint256 indexed parentId, address indexed  
childContract, uint256 indexed childTokenId);  
event Decomposed(uint256 indexed parentId, address indexed  
childContract, uint256 indexed childTokenId);
```

```
constructor() {  
    owner = msg.sender;  
}
```

```
function mint(address to) external returns (uint256) {  
    uint256 tokenId = _currentTokenId;  
    _currentTokenId ++;
```

```
    _owners[tokenId] = to;  
    _balances[to] ++;
```

```
    return tokenId;  
}
```

```
function attachChild(  
    uint256 parentTokenId,  
    address childContract,  
    uint256 childTokenId  
) external {  
    require(_owners[parentTokenId] == msg.sender);  
    require(_childToParent[childContract][childTokenId] == 0);
```

```
    IERC721(childContract).transferFrom(msg.sender, address(this),  
childTokenId);
```

```
    _compositions[parentTokenId].childTokenIds.push(childTokenId);  
    _compositions[parentTokenId].childContracts.push(childContract);  
    _childToParent[childContract][childTokenId] = parentTokenId;
```

```
    emit Composed(parentTokenId, childContract, childTokenId);  
}
```

```
function detachChild(  
    uint256 parentTokenId,  
    address childContract,
```



```

uint256 childTokenId
) external {
    require(_owners[parentTokenId] == msg.sender);
    require(_childToParent[childContract][childTokenId] ==
parentTokenId);

    TokenComposition storage comp = _compositions[parentTokenId];

    for (uint256 i = 0; i < comp.childTokenIds.length; i++) {
        if (comp.childTokenIds[i] == childTokenId &&
comp.childContracts[i] == childContract) {
            comp.childTokenIds[i] =
comp.childTokenIds[comp.childTokenIds.length - 1];
            comp.childContracts[i] =
comp.childContracts[comp.childContracts.length - 1];
            comp.childTokenIds.pop();
            comp.childContracts.pop();
            break;
        }
    }

    delete _childToParent[childContract][childTokenId];

    IERC721(childContract).transferFrom(address(this), msg.sender,
childTokenId);

    emit Decomposed(parentTokenId, childContract, childTokenId);
}

function getChildren(uint256 tokenId) external view returns (
    address[] memory contracts,
    uint256[] memory tokenIds
) {
    TokenComposition memory comp = _compositions[tokenId];
    return (comp.childContracts, comp.childTokenIds);
}

function getParent(address childContract, uint256 childTokenId)
external view returns (uint256) {
    return _childToParent[childContract][childTokenId];
}

```

```
}
```

```
function transferFrom(address from, address to, uint256 tokenId)
```

```
external {
```

```
    require(_owners[tokenId] == from);
```

```
    require(msg.sender == from || msg.sender == owner);
```

```
    _balances[from]--;
```

```
    _balances[to]++;
```

```
    _owners[tokenId] = to;
```

```
}
```

```
function ownerOf(uint256 tokenId) public view returns (address) {
```

```
    return _owners[tokenId];
```

```
}
```

```
function balanceOf(address account) public view returns (uint256)
```

```
{
```

```
    return _balances[account];
```

```
}
```

```
function onERC721Received(
```

```
    address,
```

```
    address,
```

```
    uint256,
```

```
    bytes calldata
```

```
) external pure returns (bytes4) {
```

```
    return this.onERC721Received.selector;
```

```
}
```

```
}
```

SECTION 6: RENTAL AND DELEGATION

Allow others to use your NFT without transferring ownership:

```
contract NFTRental {
```

```
    struct Rental {
```

```
        address owner;
```

```
        address renter;
```

```
uint256 pricePerDay;  
uint256 startTime;  
uint256 endTime;  
bool active;  
}
```

```
mapping(address => mapping(uint256 => Rental)) public  
rentals;  
mapping(address => mapping(uint256 => bool)) public listed;
```

```
event Listed(address indexed nftContract, uint256 indexed tokenId,  
uint256 pricePerDay);  
event Rented(address indexed nftContract, uint256 indexed  
tokenId, address indexed renter, uint256 duration);  
event RentalEnded(address indexed nftContract, uint256 indexed  
tokenId);
```

```
function listForRent(  
address nftContract,  
uint256 tokenId,  
uint256 pricePerDay  
) external {  
require(IERC721(nftContract).ownerOf(tokenId) == msg.sender);  
require(!listed[nftContract][tokenId]);
```

```
listed[nftContract][tokenId] = true;
```

```
rentals[nftContract][tokenId] = Rental({  
owner: msg.sender,  
renter: address(0),  
pricePerDay: pricePerDay,  
startTime: 0,  
endTime: 0,  
active: false  
});
```

```
emit Listed(nftContract, tokenId, pricePerDay);  
}
```

```
function rent(  

```

```

address nftContract,
uint256 tokenId,
uint256 durationDays
) external payable {
    Rental storage rental = rentals[nftContract][tokenId];

    require(listed[nftContract][tokenId]);
    require(!rental.active);
    require(msg.value >= rental.pricePerDay * durationDays);

    rental.renter = msg.sender;
    rental.startTime = block.timestamp;
    rental.endTime = block.timestamp + (durationDays * 1 days);
    rental.active = true;

    payable(rental.owner).transfer(msg.value);

    emit Rented(nftContract, tokenId, msg.sender, durationDays);
}

```

```

function endRental(address nftContract, uint256 tokenId) external
{
    Rental storage rental = rentals[nftContract][tokenId];

    require(rental.active);
    require(block.timestamp >= rental.endTime || msg.sender ==
rental.owner);

    rental.active = false;
    rental.renter = address(0);

    emit RentalEnded(nftContract, tokenId);
}

```

```

function getCurrentRenter(
address nftContract,
uint256 tokenId
) external view returns (address) {
    Rental memory rental = rentals[nftContract][tokenId];
}

```

```

if (rental.active && block.timestamp < rental.endTime) {
    return rental.renter;
}

return address(0);
}

function isRented(address nftContract, uint256 tokenId) external
view returns (bool) {
    Rental memory rental = rentals[nftContract][tokenId];
    return rental.active && block.timestamp < rental.endTime;
}
}

```

ERC-4907 Rental Standard

```

contract ERC4907 {
    struct UserInfo {
        address user;
        uint64 expires;
    }

    mapping(uint256 => UserInfo) internal _users;
    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;
    mapping(uint256 => address) private _tokenApprovals;
    mapping(address => mapping(address => bool)) private
    _operatorApprovals;

    address public owner;
    uint256 private _currentTokenId;

    event UpdateUser(uint256 indexed tokenId, address indexed user,
    uint64 expires);
    event Transfer(address indexed from, address indexed to, uint256
    indexed tokenId);
    event Approval(address indexed owner, address indexed approved,
    uint256 indexed tokenId);
    event ApprovalForAll(address indexed owner, address indexed
    operator, bool approved);
}

```

```
constructor() {  
  owner = msg.sender;  
}
```

```
function setUser(uint256 tokenId, address user, uint64 expires)  
public {  
  require(!_isApprovedOrOwner(msg.sender, tokenId));  
  UserInfo storage info = _users[tokenId];  
  info.user = user;  
  info.expires = expires;  
  emit UpdateUser(tokenId, user, expires);  
}
```

```
function userOf(uint256 tokenId) public view returns (address) {  
  if (uint256(_users[tokenId].expires) >= block.timestamp) {  
    return _users[tokenId].user;  
  }  
  return address(0);  
}
```

```
function userExpires(uint256 tokenId) public view returns  
(uint256) {  
  return _users[tokenId].expires;  
}
```

```
function mint(address to) external returns (uint256) {  
  uint256 tokenId = _currentTokenId;  
  _currentTokenId ++;
```

```
  _owners[tokenId] = to;  
  _balances[to] ++;
```

```
  emit Transfer(address(0), to, tokenId);  
  return tokenId;  
}
```

```
function ownerOf(uint256 tokenId) public view returns (address) {  
  address tokenOwner = _owners[tokenId];  
  require(tokenOwner != address(0));
```

```
return tokenOwner;  
}
```

```
function balanceOf(address account) public view returns (uint256)  
{  
    require(account != address(0));  
    return _balances[account];  
}
```

```
function transferFrom(address from, address to, uint256 tokenId)  
public {  
    require(_isApprovedOrOwner(msg.sender, tokenId));  
  
    _balances[from]--;  
    _balances[to] ++;  
    _owners[tokenId] = to;  
  
    delete _tokenApprovals[tokenId];  
    delete _users[tokenId];  
  
    emit Transfer(from, to, tokenId);  
}
```

```
function approve(address to, uint256 tokenId) public {  
    address tokenOwner = ownerOf(tokenId);  
    require(msg.sender == tokenOwner ||  
_operatorApprovals[tokenOwner][msg.sender]);  
    _tokenApprovals[tokenId] = to;  
    emit Approval(tokenOwner, to, tokenId);  
}
```

```
function setApprovalForAll(address operator, bool approved)  
public {  
    _operatorApprovals[msg.sender][operator] = approved;  
    emit ApprovalForAll(msg.sender, operator, approved);  
}
```

```
function getApproved(uint256 tokenId) public view returns  
(address) {  
    require(_owners[tokenId] != address(0));
```

```
return _tokenApprovals[tokenId];  
}
```

```
function isApprovedForAll(address tokenOwner, address operator)  
public view returns (bool) {  
    return _operatorApprovals[tokenOwner][operator];  
}
```

```
function _isApprovedOrOwner(address spender, uint256 tokenId)  
internal view returns (bool) {  
    address tokenOwner = ownerOf(tokenId);  
    return (  
        spender == tokenOwner ||  
        getApproved(tokenId) == spender ||  
        isApprovedForAll(tokenOwner, spender)  
    );  
}
```

```
function supportsInterface(bytes4 interfaceId) public pure returns  
(bool) {  
    return  
        interfaceId == 0xad092b5c ||  
        interfaceId == 0x80ac58cd ||  
        interfaceId == 0x01ffc9a7;  
}
```

SECTION 7: STAKING NFTS

Stake NFTs to earn rewards:

```
contract NFTStaking {  
    struct Stake {  
        uint256 tokenId;  
        uint256 stakedAt;  
        uint256 lastClaimed;  
    }  
}
```

```
address public nftContract;
```



```
address public rewardToken;
```

```
uint256 public rewardPerDay;
```

```
uint256 public constant PRECISION = 1e18;
```

```
mapping(address => Stake[]) public stakes;
```

```
mapping(uint256 => address) public tokenStaker;
```

```
address public owner;
```

```
event Staked(address indexed user, uint256 indexed tokenId);
```

```
event Unstaked(address indexed user, uint256 indexed tokenId);
```

```
event RewardsClaimed(address indexed user, uint256 amount);
```

```
constructor(address nft, address reward, uint256 dailyReward) {
```

```
    owner = msg.sender;
```

```
    nftContract = nft;
```

```
    rewardToken = reward;
```

```
    rewardPerDay = dailyReward;
```

```
}
```

```
function stake(uint256[] calldata tokenIds) external {
```

```
    for (uint256 i = 0; i < tokenIds.length; i++) {
```

```
        uint256 tokenId = tokenIds[i];
```

```
        IERC721(nftContract).transferFrom(msg.sender, address(this),  
tokenId);
```

```
        stakes[msg.sender].push(Stake({
```

```
            tokenId: tokenId,
```

```
            stakedAt: block.timestamp,
```

```
            lastClaimed: block.timestamp
```

```
        }));
```

```
        tokenStaker[tokenId] = msg.sender;
```

```
        emit Staked(msg.sender, tokenId);
```

```
    }
```

```
}
```

```

function unstake(uint256[] calldata tokenIds) external {
    for (uint256 i = 0; i < tokenIds.length; i++) {
        uint256 tokenId = tokenIds[i];
        require(tokenStaker[tokenId] == msg.sender);

        uint256 rewards = calculateRewards(msg.sender, tokenId);
        if (rewards > 0) {
            IERC20(rewardToken).transfer(msg.sender, rewards);
        }
    }
}

```

```

_removeStake(msg.sender, tokenId);
delete tokenStaker[tokenId];

```

```

IERC721(nftContract).transferFrom(address(this), msg.sender,
tokenId);

```

```

emit Unstaked(msg.sender, tokenId);
}
}

```

```

function claimRewards() external {
    uint256 totalRewards = 0;

    for (uint256 i = 0; i < stakes[msg.sender].length; i++) {
        Stake storage stakeInfo = stakes[msg.sender][i];
        uint256 rewards = _calculateStakeRewards(stakeInfo);
        totalRewards += rewards;
        stakeInfo.lastClaimed = block.timestamp;
    }
}

```

```

require(totalRewards > 0);
IERC20(rewardToken).transfer(msg.sender, totalRewards);

emit RewardsClaimed(msg.sender, totalRewards);
}

```

```

function calculateRewards(address user, uint256 tokenId) public
view returns (uint256) {
    for (uint256 i = 0; i < stakes[user].length; i++) {
        if (stakes[user][i].tokenId == tokenId) {

```

```

return _calculateStakeRewards(stakes[user][i]);
}
}
return 0;
}

```

```

function _calculateStakeRewards(Stake memory stakeInfo) internal
view returns (uint256) {
    uint256 duration = block.timestamp - stakeInfo.lastClaimed;
    return (duration * rewardPerDay) / 1 days;
}

```

```

function _removeStake(address user, uint256 tokenId) internal {
    Stake[] storage userStakes = stakes[user];

```

```

    for (uint256 i = 0; i < userStakes.length; i++) {
        if (userStakes[i].tokenId == tokenId) {
            userStakes[i] = userStakes[userStakes.length - 1];
            userStakes.pop();
            break;
        }
    }
}

```

```

function getStakedTokens(address user) external view returns
(uint256[] memory) {
    Stake[] memory userStakes = stakes[user];
    uint256[] memory tokenIds = new uint256[](userStakes.length);

```

```

    for (uint256 i = 0; i < userStakes.length; i++) {
        tokenIds[i] = userStakes[i].tokenId;
    }

```

```

    return tokenIds;
}

```

```

function getPendingRewards(address user) external view returns
(uint256) {
    uint256 total = 0;
    for (uint256 i = 0; i < stakes[user].length; i++) {

```

```

total += _calculateStakeRewards(stakes[user][i]);
}
return total;
}

```

```

function onERC721Received(
    address,
    address,
    uint256,
    bytes calldata
) external pure returns (bytes4) {
    return this.onERC721Received.selector;
}
}

```

```

interface IERC20 {
    function transfer(address to, uint256 amount) external returns
(bool);
}

```

SECTION 8: IMPLEMENTING MULTIPLE PATTERNS

Combining patterns creates powerful applications:

```

import { ethers } from 'ethers'

```

```

class AdvancedNFTManager {
    constructor(provider) {
        this.provider = provider
        this.contracts = {}
    }

```

```

    addContract(name, address, abi) {
        this.contracts[name] = new ethers.Contract(address, abi,
this.provider)
    }

```

```

    async checkDynamicNFTStatus(tokenId) {
        const contract = this.contracts.dynamicNFT

```

```
const attributes = await contract.attributes(tokenId)
const tokenURI = await contract.tokenURI(tokenId)

return {
  tokenId,
  level: Number(attributes.level),
  experience: Number(attributes.experience),
  strength: Number(attributes.strength),
  agility: Number(attributes.agility),
  lastUpdated: new Date(Number(attributes.lastUpdated) * 1000),
  tokenURI
}
}
```

```
async verifySoulboundCredential(tokenId) {
  const contract = this.contracts.credential
  const result = await contract.verifyCredential(tokenId)

  return {
    valid: result.valid,
    holder: result.holder,
    credentialType: result.credentialType,
    expired: result.expired
  }
}
```

```
async checkFractionOwnership(userAddress) {
  const contract = this.contracts.fractional
  const balance = await contract.balanceOf(userAddress)
  const totalSupply = await contract.totalSupply()

  return {
    balance: ethers.formatEther(balance),
    totalSupply: ethers.formatEther(totalSupply),
    ownershipPercent: (Number(balance) / Number(totalSupply) *
100).toFixed(4)
  }
}
```

```
async getLoanDetails(loanId) {
```

```
const contract = this.contracts.lending
const loan = await contract.loans(loanId)
const status = await contract.getLoanStatus(loanId)
```

```
return {
  borrower: loan.borrower,
  nftContract: loan.nftContract,
  nftTokenId: Number(loan.nftTokenId),
  loanAmount: ethers.formatEther(loan.loanAmount),
  active: status.active,
  defaulted: status.defaulted,
  totalOwed: ethers.formatEther(status.totalOwed),
  timeRemaining: Number(status.timeRemaining)
}
}
```

```
async getComposedNFTDetails(tokenId) {
  const contract = this.contracts.composable
  const [contracts, tokenIds] = await contract.getChildren(tokenId)
```

```
  const children = []
  for (let i = 0; i < contracts.length; i++) {
    children.push({
      contract: contracts[i],
      tokenId: Number(tokenIds[i])
    })
  }
}
```

```
return {
  parentTokenId: tokenId,
  owner: await contract.ownerOf(tokenId),
  children
}
}
```

```
async getRentalInfo(nftContract, tokenId) {
  const contract = this.contracts.rental
  const isRented = await contract.isRented(nftContract, tokenId)
  const currentRenter = await
  contract.getCurrentRenter(nftContract, tokenId)
```

```

const rental = await contract.rentals(nftContract, tokenId)

return {
  nftContract,
  tokenId,
  owner: rental.owner,
  isRented,
  currentRenter,
  pricePerDay: ethers.formatEther(rental.pricePerDay),
  endTime: isRented ? new Date(Number(rental.endTime) * 1000) :
null
}
}

async getStakingRewards(userAddress) {
  const contract = this.contracts.staking
  const stakedTokens = await
contract.getStakedTokens(userAddress)
  const pendingRewards = await
contract.getPendingRewards(userAddress)

  return {
    stakedCount: stakedTokens.length,
    stakedTokenIds: stakedTokens.map(Number),
    pendingRewards: ethers.formatEther(pendingRewards)
  }
}
}

```

Advanced NFT patterns transform simple ownership tokens into sophisticated digital assets. Dynamic NFTs evolve with time and interaction. Soulbound tokens create non-transferable credentials. Fractionalization democratizes expensive assets. Lending unlocks liquidity. Composability enables complex asset structures. Each pattern solves different problems -- choose based on your application's needs.

CHAPTER 8: BUILDING NFT APPLICATIONS

This chapter brings together everything from the previous seven chapters into complete, production-ready NFT applications. You

will build minting dApps, gallery displays, rarity analysis tools, collection management systems, and airdrop mechanics. These are the real applications that creators, collectors, and developers need.

SECTION 1: MINTING DAPP ARCHITECTURE

A minting dApp connects users to your NFT smart contract, handles wallet interactions, displays mint status, and provides a smooth user experience. The architecture involves frontend components, wallet connection, contract interaction, and real-time state updates.

The core components of a minting dApp are the wallet connector that interfaces with MetaMask and WalletConnect, the contract interface that calls mint functions, the state manager that tracks mint progress and supply, and the UI layer that presents everything to users.

Here is a complete React-based minting dApp structure.

This is the main minting application component.

```
import { useState, useEffect, useCallback } from 'react'
import { ethers } from 'ethers'

const NFT_CONTRACT_ADDRESS = '0x...'
const NFT_ABI = [
  'function mint(uint256 quantity) payable',
  'function totalSupply() view returns (uint256)',
  'function maxSupply() view returns (uint256)',
  'function mintPrice() view returns (uint256)',
  'function maxPerWallet() view returns (uint256)',
  'function balanceOf(address owner) view returns (uint256)',
  'function paused() view returns (bool)',
  'function whitelistMint(uint256 quantity, bytes32[] proof) payable',
  'function isWhitelisted(address user, bytes32[] proof) view returns (bool)'
]

function MintingApp() {
```



```
const [provider, setProvider] = useState(null)
const [signer, setSigner] = useState(null)
const [account, setAccount] = useState(null)
const [contract, setContract] = useState(null)
```

```
const [mintData, setMintData] = useState({
  totalSupply: 0,
  maxSupply: 0,
  mintPrice: '0',
  maxPerWallet: 0,
  userBalance: 0,
  paused: true
})
```

```
const [mintQuantity, setMintQuantity] = useState(1)
const [isMinting, setIsMinting] = useState(false)
const [txHash, setTxHash] = useState(null)
const [error, setError] = useState(null)
```

```
const connectWallet = useCallback(async () => {
  if (typeof window.ethereum === 'undefined') {
    setError('Please install MetaMask')
    return
  }
```

```
  try {
    const web3Provider = new
    ethers.BrowserProvider(window.ethereum)
    const accounts = await web3Provider.send('eth_requestAccounts',
    [])
    const web3Signer = await web3Provider.getSigner()
```

```

    const nftContract = new ethers.Contract(
      NFT_CONTRACT_ADDRESS,
      NFT_ABI,
      web3Signer
    )
```

```
    setProvider(web3Provider)
    setSigner(web3Signer)
```

```

setAccount(accounts[0])
setContract(nftContract)
setError(null)
} catch (err) {
setError('Failed to connect wallet: ' + err.message)
}
}, [])

```

```

const loadMintData = useCallback(async () => {
if (!contract || !account) return

```

```

try {
const [totalSupply, maxSupply, mintPrice, maxPerWallet,
userBalance, paused] =
await Promise.all([
contract.totalSupply(),
contract.maxSupply(),
contract.mintPrice(),
contract.maxPerWallet(),
contract.balanceOf(account),
contract.paused()
])

```

```

setMintData({
totalSupply: Number(totalSupply),
maxSupply: Number(maxSupply),
mintPrice: ethers.formatEther(mintPrice),
maxPerWallet: Number(maxPerWallet),
userBalance: Number(userBalance),
paused: paused
})
} catch (err) {
console.error('Failed to load mint data:', err)
}
}, [contract, account])

```

```

useEffect(() => {
loadMintData()
const interval = setInterval(loadMintData, 10000)
return () => clearInterval(interval)

```

```
}, [loadMintData])
```

```
useEffect(() => {  
  if (window.ethereum) {  
    window.ethereum.on('accountsChanged', (accounts) => {  
      if (accounts.length > 0) {  
        setAccount(accounts[0])  
      } else {  
        setAccount(null)  
        setContract(null)  
      }  
    })  
  }  
})
```

```
window.ethereum.on('chainChanged', () => {  
  window.location.reload()  
})  
}  
}, [])
```

```
const mint = async () => {  
  if (!contract) return
```

```
  setIsMinting(true)  
  setError(null)  
  setTxHash(null)
```

```
  try {  
    const totalCost = ethers.parseEther(mintData.mintPrice) *  
    BigInt(mintQuantity)
```

```
    const tx = await contract.mint(mintQuantity, { value: totalCost })  
    setTxHash(tx.hash)
```

```
    await tx.wait()  
    await loadMintData()
```

```
  } catch (err) {  
    if (err.code === 'ACTION_REJECTED') {  
      setError("Transaction rejected by user")  
    } else if (err.message.includes('insufficient funds')) {
```

```

setError('Insufficient ETH balance')
} else if (err.message.includes('max per wallet')) {
  setError('Exceeded maximum mints per wallet')
} else {
  setError('Mint failed: ' + err.message)
}
} finally {
  setIsMinting(false)
}
}

const remainingMints = mintData.maxPerWallet -
mintData.userBalance
const soldOut = mintData.totalSupply >= mintData.maxSupply
const canMint = !mintData.paused && !soldOut &&
remainingMints > 0

return {
  account,
  mintData,
  mintQuantity,
  setMintQuantity,
  isMinting,
  txHash,
  error,
  connectWallet,
  mint,
  remainingMints,
  soldOut,
  canMint
}
}

```

This React hook encapsulates all minting logic. The component tracks wallet connection, contract state, minting progress, and handles errors gracefully. Real-time updates refresh the mint data every 10 seconds.

SECTION 2: WHITELIST AND MERKLE PROOF INTEGRATION

Most NFT launches use whitelists to give early access to community members. The minting dApp needs to verify whitelist status and generate Merkle proofs for eligible addresses.

This module handles whitelist verification and proof generation.

```
import { MerkleTree } from 'merkletreejs'
import keccak256 from 'keccak256'

class WhitelistManager {
  constructor(whitelistAddresses) {
    this.addresses = whitelistAddresses.map(addr =>
      addr.toLowerCase())
    this.leaves = this.addresses.map(addr => keccak256(addr))
    this.tree = new MerkleTree(this.leaves, keccak256, { sortPairs:
      true })
    this.root = this.tree.getHexRoot()
  }

  isWhitelisted(address) {
    return this.addresses.includes(address.toLowerCase())
  }

  getProof(address) {
    const leaf = keccak256(address.toLowerCase())
    return this.tree.getHexProof(leaf)
  }

  verifyProof(address, proof) {
    const leaf = keccak256(address.toLowerCase())
    return this.tree.verify(proof, leaf, this.root)
  }

  async function fetchWhitelist(apiEndpoint) {
    const response = await fetch(apiEndpoint)
    const data = await response.json()
    return data.addresses
  }
```

```

class WhitelistMintingApp {
  constructor(contract, whitelistApiEndpoint) {
    this.contract = contract
    this.apiEndpoint = whitelistApiEndpoint
    this.whitelistManager = null
  }

  async initialize() {
    const addresses = await fetchWhitelist(this.apiEndpoint)
    this.whitelistManager = new WhitelistManager(addresses)
  }

  async checkEligibility(userAddress) {
    if (!this.whitelistManager) {
      await this.initialize()
    }

    const isWhitelisted =
      this.whitelistManager.isWhitelisted(userAddress)
    const proof = isWhitelisted ?
      this.whitelistManager.getProof(userAddress) : []

    let onChainVerified = false
    if (isWhitelisted) {
      try {
        onChainVerified = await this.contract.isWhitelisted(userAddress,
          proof)
      } catch (err) {
        console.error('On-chain verification failed:', err)
      }
    }

    return {
      isWhitelisted,
      proof,
      onChainVerified
    }
  }
}

```

```

async whitelistMint(quantity, userAddress, mintPrice) {
  const eligibility = await this.checkEligibility(userAddress)

  if (!eligibility.onChainVerified) {
    throw new Error('Address not whitelisted or proof invalid')
  }

  const totalCost = mintPrice * BigInt(quantity)
  const tx = await this.contract.whitelistMint(quantity,
    eligibility.proof, {
      value: totalCost
    })

  return tx
}

```

The whitelist manager builds the Merkle tree client-side and generates proofs for eligible addresses. The eligibility check verifies both locally and on-chain before allowing the mint transaction.

SECTION 3: MULTI-PHASE MINT HANDLING

Professional NFT launches have multiple phases such as OG holders, whitelist, and public sale. Each phase has different prices, limits, and timing.

This system manages multiple mint phases.

```

const MintPhase = {
  NOT_STARTED: 0,
  OG_MINT: 1,
  WHITELIST: 2,
  PUBLIC: 3,
  ENDED: 4
}

class MultiPhaseMintManager {
  constructor(contract, phaseConfig) {

```

```
this.contract = contract
this.phases = phaseConfig
}
```

```
getCurrentPhase(timestamp) {
  const now = timestamp || Math.floor(Date.now() / 1000)
```

```
  for (const phase of this.phases) {
    if (now >= phase.startTime && now < phase.endTime) {
      return phase
    }
  }
}
```

```
const lastPhase = this.phases[this.phases.length - 1]
if (now >= lastPhase.endTime) {
  return { id: MintPhase.ENDED, name: 'Ended' }
}
```

```
return { id: MintPhase.NOT_STARTED, name: 'Not Started' }
}
```

```
getTimeUntilNextPhase() {
  const now = Math.floor(Date.now() / 1000)
  const currentPhase = this.getCurrentPhase(now)
```

```
  if (currentPhase.id === MintPhase.NOT_STARTED) {
    const firstPhase = this.phases[0]
    return {
      nextPhase: firstPhase.name,
      seconds: firstPhase.startTime - now
    }
  }
}
```

```
  if (currentPhase.endTime) {
    const currentIndex = this.phases.findIndex(p => p.id ===
currentPhase.id)
    const nextPhase = this.phases[currentIndex + 1]
```

```
  return {
    nextPhase: nextPhase ? nextPhase.name : 'Ended',
```



```
seconds: currentPhase.endTime - now
}
}
```

```
return null
}
```

```
async getMintEligibility(userAddress, ogProof, wlProof) {
  const currentPhase = this.getCurrentPhase()
```

```
  const eligibility = {
    phase: currentPhase,
    canMint: false,
    maxQuantity: 0,
    price: '0',
    reason: ''
  }
```

```
  if (currentPhase.id === MintPhase.NOT_STARTED) {
    eligibility.reason = 'Mint has not started'
    return eligibility
  }
```

```
  if (currentPhase.id === MintPhase.ENDED) {
    eligibility.reason = 'Mint has ended'
    return eligibility
  }
```

```
  try {
    const userMinted = await
    this.contract.mintedByAddress(userAddress)
    const phaseMinted = await this.contract.phaseMinted(userAddress,
    currentPhase.id)
```

```
    if (currentPhase.id === MintPhase.OG_MINT) {
      const isOG = await this.contract.verifyOG(userAddress, ogProof)
      if (!isOG) {
        eligibility.reason = 'Not an OG holder'
        return eligibility
      }
    }
```

```

    eligibility.maxQuantity = currentPhase.maxPerWallet -
    Number(phaseMinted)
  }

  if (currentPhase.id === MintPhase.WHITELIST) {
    const isWL = await this.contract.verifyWhitelist(userAddress,
    wlProof)
    if (!isWL) {
      eligibility.reason = 'Not whitelisted'
      return eligibility
    }
    eligibility.maxQuantity = currentPhase.maxPerWallet -
    Number(phaseMinted)
  }

  if (currentPhase.id === MintPhase.PUBLIC) {
    eligibility.maxQuantity = currentPhase.maxPerWallet -
    Number(phaseMinted)
  }

  eligibility.canMint = eligibility.maxQuantity > 0
  eligibility.price = currentPhase.price

  if (!eligibility.canMint) {
    eligibility.reason = 'Already minted maximum for this phase'
  }

} catch (err) {
  eligibility.reason = 'Error checking eligibility: ' + err.message
}

return eligibility
}

formatCountdown(seconds) {
  const days = Math.floor(seconds / 86400)
  const hours = Math.floor((seconds % 86400) / 3600)
  const minutes = Math.floor((seconds % 3600) / 60)
  const secs = seconds % 60

```

```

if (days > 0) {
return days + 'd ' + hours + 'h ' + minutes + 'm'
}
if (hours > 0) {
return hours + 'h ' + minutes + 'm ' + secs + 's'
}
return minutes + 'm ' + secs + 's'
}
}

```

The multi-phase manager tracks which phase is active, calculates time until the next phase, and determines user eligibility based on proofs and prior mints.

SECTION 4: NFT GALLERY DISPLAY APPLICATION

An NFT gallery displays a user's collection with images, metadata, and interactive features. This requires fetching token data, resolving metadata URIs, and rendering the collection attractively.

This is a complete gallery application.

```

class NFTGallery {
  constructor(providerUrl) {
    this.provider = new ethers.JsonRpcProvider(providerUrl)
    this.metadataCache = new Map()
    this.imageCache = new Map()
  }

  async getOwnedTokens(contractAddress, ownerAddress) {
    const erc721Abi = [
      'function balanceOf(address owner) view returns (uint256)',
      'function tokenOfOwnerByIndex(address owner, uint256 index)
view returns (uint256)',
      'function tokenURI(uint256 tokenId) view returns (string)',
      'function name() view returns (string)',
      'function symbol() view returns (string)'
    ]

```

```
const contract = new ethers.Contract(contractAddress, erc721Abi,  
this.provider)
```

```
const [balance, name, symbol] = await Promise.all([  
contract.balanceOf(ownerAddress),  
contract.name(),  
contract.symbol()  
])
```

```
const tokenIds = []  
const batchSize = 50  
const totalTokens = Number(balance)
```

```
for (let i = 0; i < totalTokens; i += batchSize) {  
  const batchPromises = []  
  for (let j = i; j < Math.min(i + batchSize, totalTokens); j++) {  
    batchPromises.push(contract.tokenOfOwnerByIndex(ownerAddress,  
j))  
  }  
  const batchResults = await Promise.all(batchPromises)  
  tokenIds.push(...batchResults.map(id => Number(id)))  
}
```

```
return {  
  contractAddress,  
  name,  
  symbol,  
  tokenIds,  
  balance: totalTokens  
}  
}
```

```
async getTokenMetadata(contractAddress, tokenId) {  
  const cacheKey = contractAddress + ':' + tokenId
```

```
  if (this.metadataCache.has(cacheKey)) {  
    return this.metadataCache.get(cacheKey)  
  }
```

```
const contract = new ethers.Contract(  

```

```
contractAddress,  
['function tokenURI(uint256 tokenId) view returns (string)'],  
this.provider  
)
```

```
const tokenURI = await contract.tokenURI(tokenId)  
const resolvedURI = this.resolveIPFS(tokenURI)
```

```
const response = await fetch(resolvedURI)  
const metadata = await response.json()
```

```
metadata.image = this.resolveIPFS(metadata.image)  
if (metadata.animation_url) {  
  metadata.animation_url =  
this.resolveIPFS(metadata.animation_url)  
}
```

```
this.metadataCache.set(cacheKey, metadata)  
return metadata  
}
```

```
resolveIPFS(uri) {  
if (!uri) return uri
```

```
if (uri.startsWith('ipfs://')) {  
  return 'https://ipfs.io/ipfs/' + uri.slice(7)  
}  
if (uri.startsWith('ar://')) {  
  return 'https://arweave.net/' + uri.slice(5)  
}  
return uri  
}
```

```
async getCollectionWithMetadata(contractAddress, ownerAddress,  
options = {}) {  
  const collection = await this.getOwnedTokens(contractAddress,  
ownerAddress)
```

```
const limit = options.limit || collection.tokenIds.length  
const offset = options.offset || 0
```

```
const tokenSubset = collection.tokenIds.slice(offset, offset + limit)
```

```
const metadataPromises = tokenSubset.map(tokenId =>  
  this.getTokenMetadata(contractAddress, tokenId)  
  .then(metadata => ({ tokenId, metadata, error: null }))  
  .catch(error => ({ tokenId, metadata: null, error: error.message  
}))  
)
```

```
const results = await Promise.all(metadataPromises)
```

```
return {  
  ...collection,  
  tokens: results,  
  pagination: {  
    total: collection.tokenIds.length,  
    offset,  
    limit,  
    hasMore: offset + limit < collection.tokenIds.length  
  }  
}  
}
```

```
async getMultipleCollections(ownerAddress, contractAddresses) {  
  const collectionPromises = contractAddresses.map(address =>  
    this.getOwnedTokens(address, ownerAddress)  
    .then(data => ({ ...data, error: null }))  
    .catch(error => ({ contractAddress: address, error: error.message  
}))  
  )
```

```
  return Promise.all(collectionPromises)  
}  
}
```

```
class GalleryRenderer {  
  constructor(gallery) {  
    this.gallery = gallery  
  }  
}
```

```
renderTokenCard(token) {  
  const { tokenId, metadata } = token
```

```
  
  if (!metadata) {  
    return {  
      tokenId,  
      html: this.createErrorCard(tokenId, token.error)  
    }  
  }  
}
```

```
  
const attributes = metadata.attributes || []  
const attributeHtml = attributes.map(attr =>  
  '<div class="attribute">' +  
  '<span class="trait-type">' + attr.trait_type + '</span>' +  
  '<span class="value">' + attr.value + '</span>' +  
  '</div>'  
).join("")
```

```
  
const mediaHtml = metadata.animation_url  
  ? this.createVideoElement(metadata.animation_url,  
    metadata.image)  
  : this.createImageElement(metadata.image, metadata.name)
```

```
  
return {  
  tokenId,  
  html: '<div class="nft-card" data-token-id="' + tokenId + '">' +  
  '<div class="media-container">' + mediaHtml + '</div>' +  
  '<div class="info">' +  
  '<h3 class="name">' + (metadata.name || 'Token #' + tokenId)  
  + '</h3>' +  
  '<p class="description">' + (metadata.description || "") + '</  
p>' +  
  '<div class="attributes">' + attributeHtml + '</div>' +  
  '</div></div>'  
}  
}
```

```
  
createImageElement(src, alt) {  
  return ''
```

```

}

createVideoElement(videoSrc, posterSrc) {
  return '<video autoplay loop muted playsinline poster="' +
posterSrc + '">' +
'<source src="' + videoSrc + '" type="video/mp4" />' +
'</video>'
}

createErrorCard(tokenId, errorMessage) {
  return '<div class="nft-card error">' +
'<div class="error-message">Failed to load token #' + tokenId
+ '</div>' +
'<p>' + errorMessage + '</p></div>'
}

async renderGallery(contractAddress, ownerAddress, options = {})
{
  const collection = await this.gallery.getCollectionWithMetadata(
contractAddress,
ownerAddress,
options
)

  const cards = collection.tokens.map(token =>
this.renderTokenCard(token))

  return {
collection: {
name: collection.name,
symbol: collection.symbol,
total: collection.balance
},
cards,
pagination: collection.pagination
}
}
}

```

The gallery system fetches tokens owned by an address, resolves

IPFS metadata URIs, and renders cards with images, names, descriptions, and attributes. Batch fetching and caching optimize performance for large collections.

SECTION 5: RARITY ANALYSIS TOOLS

Rarity tools calculate how rare each NFT is based on trait distribution across the entire collection. Rarity scores help collectors identify valuable tokens.

This is a complete rarity calculation engine.

```
class RarityCalculator {
  constructor() {
    this.traitCounts = {}
    this.traitTypeWeights = {}
    this.totalSupply = 0
    this.tokenRarities = new Map()
  }

  analyzeCollection(allMetadata) {
    this.totalSupply = allMetadata.length
    this.traitCounts = {}

    for (const item of allMetadata) {
      const attributes = item.attributes || []

      for (const attr of attributes) {
        const traitType = attr.trait_type
        const value = String(attr.value)

        if (!this.traitCounts[traitType]) {
          this.traitCounts[traitType] = {}
        }

        if (!this.traitCounts[traitType][value]) {
          this.traitCounts[traitType][value] = 0
        }
      }
    }
  }
}
```

```
this.traitCounts[traitType][value] + +  
}  
}
```

```
for (const traitType of Object.keys(this.traitCounts)) {  
  const uniqueValues =  
Object.keys(this.traitCounts[traitType]).length  
  this.traitTypeWeights[traitType] = 1 / uniqueValues  
}  
}
```

```
getTraitRarity(traitType, value) {  
  if (!this.traitCounts[traitType] || !this.traitCounts[traitType]  
[value]) {  
    return { count: 0, percentage: 0, score: 0 }  
  }
```

```
  const count = this.traitCounts[traitType][value]  
  const percentage = (count / this.totalSupply) * 100  
  const score = 1 / (count / this.totalSupply)
```

```
  return { count, percentage, score }  
}
```

```
calculateTokenRarity(metadata, tokenId) {  
  const attributes = metadata.attributes || []  
  let totalScore = 0  
  const traitScores = []
```

```
  for (const attr of attributes) {  
    const rarity = this.getTraitRarity(attr.trait_type, String(attr.value))  
    const weightedScore = rarity.score *  
(this.traitTypeWeights[attr.trait_type] || 1)
```

```
    traitScores.push({  
      traitType: attr.trait_type,  
      value: attr.value,  
      count: rarity.count,  
      percentage: rarity.percentage,  
      score: rarity.score,
```

```
weightedScore  
  })
```

```
totalScore += weightedScore  
}
```

```
const traitCount = attributes.length  
const traitCountBonus = traitCount < 5 ? 1.1 : traitCount > 8 ?  
0.9 : 1  
totalScore *= traitCountBonus
```

```
const result = {  
  tokenId,  
  totalScore,  
  normalizedScore: 0,  
  rank: 0,  
  traitScores,  
  traitCount  
}
```

```
this.tokenRarities.set(tokenId, result)  
return result  
}
```

```
calculateAllRarities(allMetadata) {  
  const rarities = []
```

```
  for (let i = 0; i < allMetadata.length; i++) {  
    const metadata = allMetadata[i]  
    const tokenId = metadata.tokenId || i  
    const rarity = this.calculateTokenRarity(metadata, tokenId)  
    rarities.push(rarity)  
  }
```

```
  rarities.sort((a, b) => b.totalScore - a.totalScore)
```

```
  const maxScore = rarities[0].totalScore  
  const minScore = rarities[rarities.length - 1].totalScore
```

```
  for (let i = 0; i < rarities.length; i++) {
```

```
rarities[i].rank = i + 1
rarities[i].normalizedScore =
((rarities[i].totalScore - minScore) / (maxScore - minScore)) * 100
}
```

```
return rarities
}
```

```
getRarityTier(rank, totalSupply) {
const percentile = (rank / totalSupply) * 100
```

```
if (percentile <= 1) return { tier: 'Legendary', color: '#FFD700' }
if (percentile <= 5) return { tier: 'Epic', color: '#A020F0' }
if (percentile <= 15) return { tier: 'Rare', color: '#0080FF' }
if (percentile <= 35) return { tier: 'Uncommon', color: '#00FF00' }
return { tier: 'Common', color: '#808080' }
}
```

```
getCollectionStats() {
const stats = {
totalSupply: this.totalSupply,
traitTypes: Object.keys(this.traitCounts).length,
traits: {}
}
```

```
for (const [traitType, values] of Object.entries(this.traitCounts)) {
const valueArray = Object.entries(values)
.map(([value, count]) => ({
value,
count,
percentage: (count / this.totalSupply) * 100
}))
.sort((a, b) => a.count - b.count)
```

```
stats.traits[traitType] = {
uniqueValues: valueArray.length,
rarest: valueArray[0],
mostCommon: valueArray[valueArray.length - 1],
distribution: valueArray
}
```

```
}
```

```
return stats
```

```
}
```

```
}
```

```
class RaritySniper {
```

```
  constructor(rarityCalculator) {
```

```
    this.calculator = rarityCalculator
```

```
  }
```

```
  findUndervalued(listings, maxRank, maxPrice) {
```

```
    const opportunities = []
```

```
    for (const listing of listings) {
```

```
      const rarity = this.calculator.tokenRarities.get(listing.tokenId)
```

```
      if (!rarity) continue
```

```
      if (rarity.rank <= maxRank && parseFloat(listing.price) <= maxPrice) {
```

```
        const rarityTier = this.calculator.getRarityTier(
```

```
          rarity.rank,
```

```
          this.calculator.totalSupply
```

```
        )
```

```
        opportunities.push({
```

```
          tokenId: listing.tokenId,
```

```
          price: listing.price,
```

```
          rank: rarity.rank,
```

```
          score: rarity.normalizedScore,
```

```
          tier: rarityTier.tier,
```

```
          value: this.estimateValue(rarity.rank, listings)
```

```
        })
```

```
      }
```

```
    }
```

```
    opportunities.sort((a, b) => {
```

```
      const aRatio = a.value / parseFloat(a.price)
```

```
      const bRatio = b.value / parseFloat(b.price)
```

```
return bRatio - aRatio  
})
```

```
return opportunities  
}
```

```
estimateValue(rank, listings) {  
  const similarRankListings = listings.filter(l => {  
    const r = this.calculator.tokenRarities.get(l.tokenId)  
    return r && Math.abs(r.rank - rank) <= 50  
  })
```

```
  if (similarRankListings.length === 0) return 0
```

```
  const prices = similarRankListings.map(l => parseFloat(l.price))  
  prices.sort((a, b) => a - b)
```

```
  return prices[Math.floor(prices.length / 2)]  
}  
}
```

The rarity calculator analyzes trait distribution across the entire collection, weights traits by uniqueness, and calculates rarity scores for each token. The sniper finds undervalued listings by comparing rarity rank to asking price.

SECTION 6: COLLECTION MANAGEMENT DASHBOARD

Collection creators need tools to manage their NFT projects including viewing holder distribution, managing metadata, controlling contract settings, and monitoring sales.

This is a collection management system.

```
class CollectionManager {  
  constructor(contractAddress, signer) {  
    this.contractAddress = contractAddress  
    this.signer = signer
```

```

this.adminAbi = [
'function owner() view returns (address)',
'function paused() view returns (bool)',
'function setPaused(bool _paused)',
'function setBaseURI(string _baseURI)',
'function setMintPrice(uint256 _price)',
'function setMaxPerWallet(uint256 _max)',
'function withdraw()',
'function withdrawTo(address _to)',
'function setRoyaltyInfo(address _receiver, uint96 _feeNumerator)',
'function totalSupply() view returns (uint256)',
'function maxSupply() view returns (uint256)',
'function mintPrice() view returns (uint256)',
'function baseURI() view returns (string)',
'event Transfer(address indexed from, address indexed to, uint256
indexed tokenId)'
]

```

```

this.contract = new ethers.Contract(contractAddress,
this.adminAbi, signer)
}

```

```

async getContractState() {
const [owner, paused, totalSupply, maxSupply, mintPrice] = await
Promise.all([
this.contract.owner(),
this.contract.paused(),
this.contract.totalSupply(),
this.contract.maxSupply(),
this.contract.mintPrice()
])
}

```

```

const balance = await
this.signer.provider.getBalance(this.contractAddress)

```

```

return {
owner,
paused,
totalSupply: Number(totalSupply),
maxSupply: Number(maxSupply),

```

```
mintPrice: ethers.formatEther(mintPrice),  
contractBalance: ethers.formatEther(balance),  
mintPercentage: (Number(totalSupply) / Number(maxSupply)) *  
100  
}  
}
```

```
async setPaused(paused) {  
  const tx = await this.contract.setPaused(paused)  
  await tx.wait()  
  return tx.hash  
}
```

```
async setBaseURI(newBaseURI) {  
  const tx = await this.contract.setBaseURI(newBaseURI)  
  await tx.wait()  
  return tx.hash  
}
```

```
async setMintPrice(priceInEth) {  
  const priceWei = ethers.parseEther(priceInEth)  
  const tx = await this.contract.setMintPrice(priceWei)  
  await tx.wait()  
  return tx.hash  
}
```

```
async withdraw(toAddress = null) {  
  const tx = toAddress  
  ? await this.contract.withdrawTo(toAddress)  
  : await this.contract.withdraw()  
  await tx.wait()  
  return tx.hash  
}
```

```
async setRoyalties(receiverAddress, percentageBasisPoints) {  
  const tx = await this.contract.setRoyaltyInfo(receiverAddress,  
percentageBasisPoints)  
  await tx.wait()  
  return tx.hash  
}
```



```

async getHolderDistribution() {
  const filter = this.contract.filters.Transfer()
  const events = await this.contract.queryFilter(filter, 0, 'latest')

  const ownership = new Map()

  for (const event of events) {
    const from = event.args.from
    const to = event.args.to
    const tokenId = Number(event.args.tokenId)

    if (from !== ethers.ZeroAddress) {
      const fromTokens = ownership.get(from) || new Set()
      fromTokens.delete(tokenId)
      if (fromTokens.size === 0) {
        ownership.delete(from)
      } else {
        ownership.set(from, fromTokens)
      }
    }

    const toTokens = ownership.get(to) || new Set()
    toTokens.add(tokenId)
    ownership.set(to, toTokens)
  }

  const holders = []
  for (const [address, tokens] of ownership) {
    if (address !== ethers.ZeroAddress) {
      holders.push({
        address,
        tokenCount: tokens.size,
        tokenIds: Array.from(tokens)
      })
    }
  }

  holders.sort((a, b) => b.tokenCount - a.tokenCount)

```

```
const totalSupply = holders.reduce((sum, h) => sum +  
h.tokenCount, 0)  
const uniqueHolders = holders.length
```

```
const distribution = {  
'1': 0,  
'2-5': 0,  
'6-10': 0,  
'11-25': 0,  
'26-50': 0,  
'51 +': 0  
}
```

```
for (const holder of holders) {  
  if (holder.tokenCount === 1) distribution['1'] ++  
  else if (holder.tokenCount <= 5) distribution['2-5'] ++  
  else if (holder.tokenCount <= 10) distribution['6-10'] ++  
  else if (holder.tokenCount <= 25) distribution['11-25'] ++  
  else if (holder.tokenCount <= 50) distribution['26-50'] ++  
  else distribution['51 +'] ++  
}
```

```
return {  
  totalSupply,  
  uniqueHolders,  
  averageHolding: totalSupply / uniqueHolders,  
  topHolders: holders.slice(0, 20),  
  distribution,  
  allHolders: holders  
}  
}
```

```
async getMintHistory() {  
  const filter = this.contract.filters.Transfer(ethers.ZeroAddress)  
  const events = await this.contract.queryFilter(filter, 0, 'latest')
```

```
  const mints = []  
  for (const event of events) {  
    const block = await event.getBlock()
```

```

mints.push({
  tokenId: Number(event.args.tokenId),
  minter: event.args.to,
  timestamp: block.timestamp,
  blockNumber: event.blockNumber,
  transactionHash: event.transactionHash
})
}

```

```

return mints.sort((a, b) => a.timestamp - b.timestamp)
}

```

```

async getRevenueStats() {
  const state = await this.getContractState()
  const mintHistory = await this.getMintHistory()

```

```

  const mintPrice = parseFloat(state.mintPrice)
  const totalMinted = state.totalSupply
  const grossRevenue = mintPrice * totalMinted

```

```

  const dailyMints = {}
  for (const mint of mintHistory) {
    const date = new Date(mint.timestamp *
1000).toISOString().split("T")[0]
    if (!dailyMints[date]) {
      dailyMints[date] = 0
    }
    dailyMints[date] + +
  }
}

```

```

return {
  grossRevenue,
  totalMinted,
  mintPrice,
  contractBalance: parseFloat(state.contractBalance),
  dailyMints
}
}
}

```

The collection manager provides administrative control over contract settings, holder analytics, mint history, and revenue tracking. Holder distribution analysis shows how tokens are spread across wallets.

SECTION 7: AIRDROP SYSTEM

Airdrops distribute NFTs to multiple addresses efficiently. This requires careful gas optimization, batch processing, and tracking which addresses have received tokens.

This is a complete airdrop system.

The smart contract for efficient batch airdrops.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract AirdropNFT is ERC721Enumerable, Ownable {
    uint256 private _tokenIdCounter;
    uint256 public maxSupply;
    string private _baseTokenURI;

    mapping(address => bool) public hasReceivedAirdrop;

    constructor(
        string memory name,
        string memory symbol,
        uint256 _maxSupply,
        string memory baseURI
    ) ERC721(name, symbol) Ownable(msg.sender) {
        maxSupply = _maxSupply;
        _baseTokenURI = baseURI;
    }
}
```

```
function airdrop(address[] calldata recipients) external onlyOwner
{
    uint256 count = recipients.length;
    require(_tokenIdCounter + count <= maxSupply, "Would exceed
max supply");
```

```
    for (uint256 i = 0; i < count; i++) {
        address recipient = recipients[i];
```

```
        if (!hasReceivedAirdrop[recipient] && recipient != address(0)) {
            uint256 tokenId = _tokenIdCounter;
            _tokenIdCounter++;
            _safeMint(recipient, tokenId);
            hasReceivedAirdrop[recipient] = true;
        }
    }
}
```

```
function airdropWithIds(
    address[] calldata recipients,
    uint256[] calldata tokenIds
) external onlyOwner {
    require(recipients.length == tokenIds.length, "Array length
mismatch");
```

```
    for (uint256 i = 0; i < recipients.length; i++) {
        require(tokenIds[i] < maxSupply, "Token ID exceeds max");
        _safeMint(recipients[i], tokenIds[i]);
    }
}
```

```
function airdropQuantity(
    address[] calldata recipients,
    uint256[] calldata quantities
) external onlyOwner {
    require(recipients.length == quantities.length, "Array length
mismatch");
```

```
    for (uint256 i = 0; i < recipients.length; i++) {
        for (uint256 j = 0; j < quantities[i]; j++) {
```

```

require(_tokenIdCounter < maxSupply, "Would exceed max
supply");
_safeMint(recipients[i], _tokenIdCounter);
_tokenIdCounter + +;
}
}
}

```

```

function setBaseURI(string calldata baseURI) external onlyOwner {
_baseTokenURI = baseURI;
}

```

```

function _baseURI() internal view override returns (string memory)
{
return _baseTokenURI;
}
}

```

The JavaScript client for managing airdrops.

```

class AirdropManager {
constructor(contractAddress, signer) {
this.contractAddress = contractAddress
this.signer = signer

this.abi = [
'function airdrop(address[] recipients)',
'function airdropWithIds(address[] recipients, uint256[] tokenIds)',
'function airdropQuantity(address[] recipients, uint256[]
quantities)',
'function hasReceivedAirdrop(address) view returns (bool)',
'function totalSupply() view returns (uint256)',
'function maxSupply() view returns (uint256)'
]

this.contract = new ethers.Contract(contractAddress, this.abi,
signer)
}

async validateAddresses(addresses) {

```

```

const valid = []
const invalid = []
const duplicates = []
const alreadyAirdropped = []

const seen = new Set()

for (const address of addresses) {
  if (!ethers.isAddress(address)) {
    invalid.push({ address, reason: 'Invalid address format' })
    continue
  }

  const checksummed = ethers.getAddress(address)

  if (seen.has(checksummed)) {
    duplicates.push(checksummed)
    continue
  }
  seen.add(checksummed)

  const received = await
  this.contract.hasReceivedAirdrop(checksummed)
  if (received) {
    alreadyAirdropped.push(checksummed)
    continue
  }

  valid.push(checksummed)
}

return { valid, invalid, duplicates, alreadyAirdropped }
}

async estimateGas(recipients) {
  try {
    const gasEstimate = await
    this.contract.airdrop.estimateGas(recipients)
    const feeData = await this.signer.provider.getFeeData()
  }
}

```

```
const gasCost = gasEstimate * feeData.gasPrice
```

```
return {  
  gasLimit: gasEstimate,  
  gasPrice: ethers.formatGwei(feeData.gasPrice) + ' gwei',  
  estimatedCost: ethers.formatEther(gasCost) + ' ETH',  
  perRecipient: ethers.formatEther(gasCost /  
    BigInt(recipients.length)) + ' ETH'  
}  
} catch (err) {  
  return { error: err.message }  
}  
}
```

```
async executeAirdrop(addresses, batchSize = 100, onProgress =  
null) {  
  const validation = await this.validateAddresses(addresses)
```

```
  if (validation.valid.length === 0) {  
    return {  
      success: false,  
      error: 'No valid addresses to airdrop',  
      validation  
    }  
  }  
}
```

```
const results = {  
  successful: [],  
  failed: [],  
  transactions: [],  
  validation  
}
```

```
const batches = []  
for (let i = 0; i < validation.valid.length; i += batchSize) {  
  batches.push(validation.valid.slice(i, i + batchSize))  
}
```

```
for (let i = 0; i < batches.length; i++) {  
  const batch = batches[i]
```



```
if (onProgress) {  
  onProgress({  
    currentBatch: i + 1,  
    totalBatches: batches.length,  
    addresses: batch.length,  
    status: 'processing'  
  })  
}
```

```
try {  
  const gasEstimate = await this.contract.airdrop.estimateGas(batch)  
  const gasLimit = (gasEstimate * 120n) / 100n
```

```
  const tx = await this.contract.airdrop(batch, { gasLimit })  
  const receipt = await tx.wait()
```

```
  results.transactions.push({  
    batch: i + 1,  
    hash: tx.hash,  
    gasUsed: receipt.gasUsed.toString(),  
    recipients: batch.length  
  })
```

```
  results.successful.push(...batch)
```

```
} catch (err) {  
  results.failed.push({  
    batch: i + 1,  
    addresses: batch,  
    error: err.message  
  })  
}  
}
```

```
results.success = results.failed.length === 0  
results.summary = {  
  total: addresses.length,  
  successful: results.successful.length,  
  failed: results.failed.reduce((sum, f) => sum + f.addresses.length,
```

```
0),
  skipped: validation.invalid.length + validation.duplicates.length +
validation.alreadyAirdropped.length
}
```

```
return results
}
```

```
async getAirdropStatus(addresses) {
  const status = []
```

```
  const batchSize = 100
  for (let i = 0; i < addresses.length; i += batchSize) {
    const batch = addresses.slice(i, i + batchSize)
    const checks = await Promise.all(
      batch.map(async addr => ({
        address: addr,
        received: await this.contract.hasReceivedAirdrop(addr)
      }))
    )
    status.push(...checks)
  }
```

```
  return status
}
}
```

```
class AirdropListBuilder {
  constructor() {
    this.addresses = new Set()
  }
```

```
  addFromCSV(csvContent) {
    const lines = csvContent.split('\n')
    for (const line of lines) {
      const address = line.split(',')[0].trim()
      if (ethers.isAddress(address)) {
        this.addresses.add(ethers.getAddress(address))
      }
    }
  }
```

```
return this.addresses.size
}
```

```
addFromHolderSnapshot(holderData) {
  for (const holder of holderData) {
    if (ethers.isAddress(holder.address)) {
      this.addresses.add(ethers.getAddress(holder.address))
    }
  }
  return this.addresses.size
}
```

```
async addFromContractHolders(contractAddress, provider) {
  const erc721Abi = [
    'function totalSupply() view returns (uint256)',
    'function ownerOf(uint256 tokenId) view returns (address)',
    'event Transfer(address indexed from, address indexed to, uint256 indexed tokenId)'
  ]
```

```
const contract = new ethers.Contract(contractAddress, erc721Abi,
provider)
```

```
const filter = contract.filters.Transfer()
const events = await contract.queryFilter(filter, 0, 'latest')
```

```
const currentOwners = new Map()
for (const event of events) {
  const tokenId = event.args.tokenId.toString()
  currentOwners.set(tokenId, event.args.to)
}
```

```
for (const owner of currentOwners.values()) {
  if (owner !== ethers.ZeroAddress) {
    this.addresses.add(ethers.getAddress(owner))
  }
}
```

```
return this.addresses.size
}
```

```

excludeAddresses(addressesToExclude) {
  for (const addr of addressesToExclude) {
    if (ethers.isAddress(addr)) {
      this.addresses.delete(ethers.getAddress(addr))
    }
  }
  return this.addresses.size
}

```

```

getList() {
  return Array.from(this.addresses)
}

```

```

exportCSV() {
  return this.getList().join('\n')
}
}

```

The airdrop system validates addresses, batches transactions for gas efficiency, tracks progress, and handles failures gracefully. The list builder aggregates addresses from multiple sources including CSV files and existing holder snapshots.

SECTION 8: REVEAL MECHANICS IMPLEMENTATION

Many NFT collections launch with hidden metadata that reveals after minting completes. This creates excitement and prevents rarity sniping during mint.

This is a complete reveal system.

```

class RevealManager {
  constructor(contractAddress, signer, pinataApiKey, pinataSecret) {
    this.contractAddress = contractAddress
    this.signer = signer
    this.pinataApiKey = pinataApiKey
    this.pinataSecret = pinataSecret

    this.abi = [

```

```

'function setBaseURI(string _baseURI)',
'function setRevealed(bool _revealed)',
'function revealed() view returns (bool)',
'function baseURI() view returns (string)',
'function totalSupply() view returns (uint256)'
]

```

```

this.contract = new ethers.Contract(contractAddress, this.abi,
signer)
}

```

```

async createPlaceholderMetadata(totalSupply,
placeholderImageCID) {
  const placeholder = {
    name: 'Unrevealed',
    description: 'This NFT has not been revealed yet. Check back
soon!',
    image: 'ipfs://' + placeholderImageCID,
    attributes: []
  }
}

```

```

const metadataFiles = []
for (let i = 0; i < totalSupply; i++) {
  metadataFiles.push({
    tokenId: i,
    content: JSON.stringify(placeholder)
  })
}

```

```

return metadataFiles
}

```

```

async shuffleAndAssignMetadata(originalMetadata, seed) {
  const shuffled = [...originalMetadata]

```

```

  let seedNum = 0
  for (let i = 0; i < seed.length; i++) {
    seedNum += seed.charCodeAt(i)
  }
}

```

```
const random = (max) => {  
  seedNum = (seedNum * 1103515245 + 12345) % 2147483648  
  return seedNum % max  
}
```

```
for (let i = shuffled.length - 1; i > 0; i--) {  
  const j = random(i + 1)  
  const temp = shuffled[i]  
  shuffled[i] = shuffled[j]  
  shuffled[j] = temp  
}
```

```
const assigned = shuffled.map((metadata, index) => ({  
  tokenId: index,  
  ...metadata  
})))
```

```
return assigned  
}
```

```
async uploadToIPFS(files) {  
  const formData = new FormData()
```

```
  for (const file of files) {  
    const blob = new Blob([file.content], { type: 'application/json' })  
    formData.append('file', blob, file.tokenId.toString())  
  }
```

```
  const response = await fetch('https://api.pinata.cloud/pinning/  
pinFileToIPFS', {  
    method: 'POST',  
    headers: {  
      'pinata_api_key': this.pinataApiKey,  
      'pinata_secret_api_key': this.pinataSecret  
    },  
    body: formData  
  })
```

```
  const result = await response.json()  
  return result.IpfsHash
```

```
}
```

```
async prepareReveal(revealedMetadata) {  
  const files = revealedMetadata.map(item => ({  
    tokenId: item.tokenId,  
    content: JSON.stringify({  
      name: item.name,  
      description: item.description,  
      image: item.image,  
      attributes: item.attributes,  
      animation_url: item.animation_url  
    })  
  })  
})
```

```
const cid = await this.uploadToIPFS(files)  
return 'ipfs://' + cid + '/'  
}
```

```
async executeReveal(newBaseURI) {  
  const currentlyRevealed = await this.contract.revealed()
```

```
  if (currentlyRevealed) {  
    return { success: false, error: 'Collection already revealed' }  
  }
```

```
  const setURITx = await this.contract.setBaseURI(newBaseURI)  
  await setURITx.wait()
```

```
  const revealTx = await this.contract.setRevealed(true)  
  await revealTx.wait()
```

```
  return {  
    success: true,  
    baseURI: newBaseURI,  
    transactions: [setURITx.hash, revealTx.hash]  
  }  
}
```

```
async scheduleReveal(revealTimestamp, newBaseURI) {  
  const now = Math.floor(Date.now() / 1000)
```

```
const delay = (revealTimestamp - now) * 1000
```

```
if (delay <= 0) {  
  return this.executeReveal(newBaseURI)  
}
```

```
return new Promise((resolve) => {  
  setTimeout(async () => {  
    const result = await this.executeReveal(newBaseURI)  
    resolve(result)  
  }, delay)  
})  
}  
}
```

```
class DelayedRevealContract {  
  static getContractCode() {  
    return `  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";  
import "@openzeppelin/contracts/access/Ownable.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";
```

```
contract DelayedRevealNFT is ERC721, Ownable {  
  using Strings for uint256;
```

```
  bool public revealed;  
  string private _revealedBaseURI;  
  string private _unrevealedURI;  
  uint256 public revealTimestamp;  
  bytes32 private _provenanceHash;
```

```
  constructor(  
    string memory name,  
    string memory symbol,  
    string memory unrevealedURI,  
    bytes32 provenanceHash  
  ) ERC721(name, symbol) Ownable(msg.sender) {
```



```

_unrevealedURI = unrevealedURI;
_provenanceHash = provenanceHash;
}

function tokenURI(uint256 tokenId) public view override returns
(string memory) {
    require(_ownerOf(tokenId) != address(0), "Token does not exist");

    if (!revealed) {
        return _unrevealedURI;
    }

    return string(abi.encodePacked(_revealedBaseURI,
tokenId.toString()));
}

function reveal(string calldata revealedBaseURI) external
onlyOwner {
    require(!revealed, "Already revealed");
    revealed = true;
    _revealedBaseURI = revealedBaseURI;
    revealTimestamp = block.timestamp;
}

function provenanceHash() external view returns (bytes32) {
    return _provenanceHash;
}
}
}
}
}

```

The reveal manager handles placeholder metadata, shuffles revealed metadata with deterministic randomness, uploads to IPFS, and executes the reveal transaction. The provenance hash proves the metadata ordering was determined before reveal.

SECTION 9: COMPLETE MINTING SITE TEMPLATE

This section provides a complete minting site structure combining all previous components.

The main application entry point.

```
import { useState, useEffect } from 'react'
import { ethers } from 'ethers'

const CONTRACT_ADDRESS = '0x...'
const CONTRACT_ABI = [
  'function mint(uint256 quantity) payable',
  'function whitelistMint(uint256 quantity, bytes32[] proof) payable',
  'function totalSupply() view returns (uint256)',
  'function maxSupply() view returns (uint256)',
  'function mintPrice() view returns (uint256)',
  'function wlMintPrice() view returns (uint256)',
  'function maxPerWallet() view returns (uint256)',
  'function balanceOf(address) view returns (uint256)',
  'function mintPhase() view returns (uint8)',
  'function paused() view returns (bool)'
]

const MINT_PHASES = {
  0: 'Not Started',
  1: 'Whitelist',
  2: 'Public',
  3: 'Ended'
}

class MintingSite {
  constructor() {
    this.provider = null
    this.signer = null
    this.contract = null
    this.account = null
    this.chainId = null
    this.listeners = new Set()
  }

  subscribe(callback) {
```

```
this.listeners.add(callback)
return () => this.listeners.delete(callback)
}
```

```
notify(state) {
  for (const listener of this.listeners) {
    listener(state)
  }
}
```

```
async connect() {
  if (!window.ethereum) {
    throw new Error('Please install MetaMask')
  }
```

```
this.provider = new ethers.BrowserProvider(window.ethereum)
const network = await this.provider.getNetwork()
this.chainId = Number(network.chainId)
```

```
const accounts = await this.provider.send('eth_requestAccounts',
[])
this.account = accounts[0]
```

```
this.signer = await this.provider.getSigner()
this.contract = new ethers.Contract(CONTRACT_ADDRESS,
CONTRACT_ABI, this.signer)
```

```
window.ethereum.on('accountsChanged',
this.handleAccountChange.bind(this))
window.ethereum.on('chainChanged', () =>
window.location.reload())
```

```
const state = await this.getState()
this.notify(state)
```

```
return state
}
```

```
handleAccountChange(accounts) {
  if (accounts.length === 0) {
```

```
this.account = null
this.notify({ connected: false })
} else {
this.account = accounts[0]
this.getState().then(state => this.notify(state))
}
}
```

```
async getState() {
if (!this.contract || !this.account) {
return { connected: false }
}
```

```
const [
totalSupply,
maxSupply,
mintPrice,
wlMintPrice,
maxPerWallet,
userBalance,
mintPhase,
paused
] = await Promise.all([
this.contract.totalSupply(),
this.contract.maxSupply(),
this.contract.mintPrice(),
this.contract.wlMintPrice(),
this.contract.maxPerWallet(),
this.contract.balanceOf(this.account),
this.contract.mintPhase(),
this.contract.paused()
])
```

```
const ethBalance = await this.provider.getBalance(this.account)
```

```
return {
connected: true,
account: this.account,
chainId: this.chainId,
totalSupply: Number(totalSupply),
```

```

maxSupply: Number(maxSupply),
mintPrice: ethers.formatEther(mintPrice),
wlMintPrice: ethers.formatEther(wlMintPrice),
maxPerWallet: Number(maxPerWallet),
userBalance: Number(userBalance),
mintPhase: Number(mintPhase),
mintPhaseName: MINT_PHASES[Number(mintPhase)],
paused,
ethBalance: ethers.formatEther(ethBalance),
remainingMints: Number(maxPerWallet) - Number(userBalance),
soldOut: Number(totalSupply) >= Number(maxSupply),
supplyRemaining: Number(maxSupply) - Number(totalSupply)
}
}

```

```

async mint(quantity, merkleProof = null) {
  if (!this.contract) {
    throw new Error('Not connected')
  }
}

```

```

const state = await this.getState()

```

```

if (state.paused) {
  throw new Error('Minting is paused')
}

```

```

if (state.soldOut) {
  throw new Error('Sold out')
}

```

```

if (quantity > state.remainingMints) {
  throw new Error('Exceeds remaining mints for your wallet')
}

```

```

let tx
if (state.mintPhase === 1 && merkleProof) {
  const price = ethers.parseEther(state.wlMintPrice) *
  BigInt(quantity)
  tx = await this.contract.whitelistMint(quantity, merkleProof, {
    value: price })
}

```

```
} else {  
  const price = ethers.parseEther(state.mintPrice) * BigInt(quantity)  
  tx = await this.contract.mint(quantity, { value: price })  
}
```

```
const receipt = await tx.wait()
```

```
const newState = await this.getState()  
this.notify(newState)
```

```
const mintedTokenIds = []  
for (const log of receipt.logs) {  
  try {  
    const parsed = this.contract.interface.parseLog(log)  
    if (parsed && parsed.name === 'Transfer') {  
      mintedTokenIds.push(Number(parsed.args.tokenId))  
    }  
  } catch (e) {}  
}
```

```
return {  
  success: true,  
  transactionHash: tx.hash,  
  tokenIds: mintedTokenIds,  
  gasUsed: receipt.gasUsed.toString()  
}  
}
```

```
disconnect() {  
  this.provider = null  
  this.signer = null  
  this.contract = null  
  this.account = null  
  this.notify({ connected: false })  
}  
}
```

```
function useMintingSite() {  
  const [site] = useState(() => new MintingSite())  
  const [state, setState] = useState({ connected: false })
```

```
const [minting, setMinting] = useState(false)
const [error, setError] = useState(null)
const [success, setSuccess] = useState(null)
```

```
useEffect(() => {
  return site.subscribe(setState)
}, [site])
```

```
const connect = async () => {
  setError(null)
  try {
    await site.connect()
  } catch (err) {
    setError(err.message)
  }
}
```

```
const mint = async (quantity, proof) => {
  setMinting(true)
  setError(null)
  setSuccess(null)
```

```
  try {
    const result = await site.mint(quantity, proof)
    setSuccess(result)
  } catch (err) {
    setError(err.message)
  } finally {
    setMinting(false)
  }
}
```

```
const refresh = async () => {
  if (site.contract) {
    const newState = await site.getState()
    setState(newState)
  }
}
```

```
return {
```

```

state,
minting,
error,
success,
connect,
mint,
refresh,
disconnect: () => site.disconnect()
}
}

```

This complete minting site template provides wallet connection, state management, multi-phase support, error handling, and a React hook for easy integration into any frontend framework.

SECTION 10: ANALYTICS AND MONITORING

Production NFT applications need analytics to track mints, sales, and user behavior.

This is a comprehensive analytics system.

```

class NFTAnalytics {
  constructor(contractAddress, provider, startBlock = 0) {
    this.contractAddress = contractAddress
    this.provider = provider
    this.startBlock = startBlock

    this.abi = [
      'event Transfer(address indexed from, address indexed to, uint256 indexed tokenId)',
      'event Approval(address indexed owner, address indexed approved, uint256 indexed tokenId)',
      'function totalSupply() view returns (uint256)',
      'function maxSupply() view returns (uint256)'
    ]

    this.contract = new ethers.Contract(contractAddress, this.abi, provider)
  }
}

```



```
}
```

```
async getMintAnalytics() {  
  const filter = this.contract.filters.Transfer(ethers.ZeroAddress)  
  const events = await this.contract.queryFilter(filter,  
this.startBlock, 'latest')
```

```
  
  const mints = []  
  const mintsByHour = {}  
  const mintsByDay = {}  
  const minterCounts = {}
```

```
  
  for (const event of events) {  
    const block = await event.getBlock()  
    const timestamp = block.timestamp  
    const date = new Date(timestamp * 1000)
```

```
  
    const hourKey = date.toISOString().slice(0, 13)  
    const dayKey = date.toISOString().slice(0, 10)
```

```
  
    mintsByHour[hourKey] = (mintsByHour[hourKey] || 0) + 1  
    mintsByDay[dayKey] = (mintsByDay[dayKey] || 0) + 1
```

```
  
    const minter = event.args.to  
    minterCounts[minter] = (minterCounts[minter] || 0) + 1
```

```
  
    mints.push({  
      tokenId: Number(event.args.tokenId),  
      minter,  
      timestamp,  
      blockNumber: event.blockNumber,  
      transactionHash: event.transactionHash  
    })  
  }  
}
```

```
  
const uniqueMinters = Object.keys(minterCounts).length  
const totalMints = mints.length
```

```
  
const topMinters = Object.entries(minterCounts)  
  .map(([address, count]) => ({ address, count })))
```

```
.sort((a, b) => b.count - a.count)
.slice(0, 10)
```

```
let peakHour = null
let peakHourMints = 0
for (const [hour, count] of Object.entries(mintsByHour)) {
  if (count > peakHourMints) {
    peakHour = hour
    peakHourMints = count
  }
}
```

```
const firstMint = mints[0]
const lastMint = mints[mints.length - 1]
const mintDuration = lastMint ? lastMint.timestamp -
firstMint.timestamp : 0
const avgMintsPerHour = mintDuration > 0
? totalMints / (mintDuration / 3600)
: totalMints
```

```
return {
  totalMints,
  uniqueMinters,
  averageMintsPerWallet: totalMints / uniqueMinters,
  mintsByHour,
  mintsByDay,
  topMinters,
  peakHour,
  peakHourMints,
  avgMintsPerHour,
  firstMintTimestamp: firstMint?.timestamp,
  lastMintTimestamp: lastMint?.timestamp,
  mintDurationHours: mintDuration / 3600
}
```

```
async getTransferAnalytics() {
  const filter = this.contract.filters.Transfer()
  const events = await this.contract.queryFilter(filter,
this.startBlock, 'latest')
```

```

const secondarySales = []
const tokenTransferCounts = {}

for (const event of events) {
  if (event.args.from === ethers.ZeroAddress) continue

  const tokenId = Number(event.args.tokenId)
  tokenTransferCounts[tokenId] = (tokenTransferCounts[tokenId] ||
0) + 1

  const block = await event.getBlock()

  secondarySales.push({
    tokenId,
    from: event.args.from,
    to: event.args.to,
    timestamp: block.timestamp,
    transactionHash: event.transactionHash
  })
}

const mostTraded = Object.entries(tokenTransferCounts)
  .map(([tokenId, transfers]) => ({ tokenId: Number(tokenId),
transfers })))
  .sort((a, b) => b.transfers - a.transfers)
  .slice(0, 10)

return {
  totalTransfers: secondarySales.length,
  uniqueTokensTraded: Object.keys(tokenTransferCounts).length,
  mostTraded,
  averageTransfersPerToken: secondarySales.length /
Object.keys(tokenTransferCounts).length,
  recentTransfers: secondarySales.slice(-20).reverse()
}
}

async getCurrentHolderStats() {
  const filter = this.contract.filters.Transfer()

```

```
const events = await this.contract.queryFilter(filter,  
this.startBlock, 'latest')
```

```
const ownership = new Map()
```

```
for (const event of events) {  
  const from = event.args.from  
  const to = event.args.to  
  const tokenId = Number(event.args.tokenId)
```

```
  if (from !== ethers.ZeroAddress) {  
    const tokens = ownership.get(from) || new Set()  
    tokens.delete(tokenId)  
    if (tokens.size === 0) {  
      ownership.delete(from)  
    } else {  
      ownership.set(from, tokens)  
    }  
  }  
}
```

```
const tokens = ownership.get(to) || new Set()  
tokens.add(tokenId)  
ownership.set(to, tokens)  
}
```

```
ownership.delete(ethers.ZeroAddress)
```

```
const holders = []  
for (const [address, tokens] of ownership) {  
  holders.push({  
    address,  
    count: tokens.size  
  })  
}
```

```
holders.sort((a, b) => b.count - a.count)
```

```
const totalSupply = holders.reduce((sum, h) => sum + h.count,  
0)  
const uniqueHolders = holders.length
```

```
const top10Holdings = holders.slice(0, 10).reduce((sum, h) =>
sum + h.count, 0)
const concentrationTop10 = (top10Holdings / totalSupply) * 100
```

```
const holdingDistribution = { 1: 0, '2-5': 0, '6-10': 0, '11-25': 0,
'26+': 0 }
for (const holder of holders) {
  if (holder.count === 1) holdingDistribution['1']++
  else if (holder.count <= 5) holdingDistribution['2-5']++
  else if (holder.count <= 10) holdingDistribution['6-10']++
  else if (holder.count <= 25) holdingDistribution['11-25']++
  else holdingDistribution['26+']++
}
```

```
return {
  totalSupply,
  uniqueHolders,
  averageHolding: totalSupply / uniqueHolders,
  topHolders: holders.slice(0, 20),
  concentrationTop10,
  holdingDistribution
}
}
```

```
async generateReport() {
  const [mintAnalytics, transferAnalytics, holderStats] = await
  Promise.all([
    this.getMintAnalytics(),
    this.getTransferAnalytics(),
    this.getCurrentHolderStats()
  ])
}
```

```
const [totalSupply, maxSupply] = await Promise.all([
  this.contract.totalSupply(),
  this.contract.maxSupply()
])
```

```
return {
  collection: {
```

```

address: this.contractAddress,
totalSupply: Number(totalSupply),
maxSupply: Number(maxSupply),
mintPercentage: (Number(totalSupply) / Number(maxSupply)) *
100
},
minting: mintAnalytics,
trading: transferAnalytics,
holders: holderStats,
generatedAt: new Date().toISOString()
}
}
}

```

The analytics system tracks mint patterns, secondary sales, holder distribution, and concentration metrics. The comprehensive report provides a complete picture of collection health.

SECTION 11: PUTTING IT ALL TOGETHER

This final section shows how to combine all components into a production NFT application.

```

class NFTApplicationSuite {
  constructor(config) {
    this.config = config
    this.provider = new ethers.JsonRpcProvider(config.rpcUrl)
    this.initialized = false
  }

  async initialize(signer) {
    this.signer = signer

    this.gallery = new NFTGallery(this.config.rpcUrl)

    this.rarityCalculator = new RarityCalculator()

    this.airdropManager = new AirdropManager(
      this.config.contractAddress,

```

```
signer  
)
```

```
this.collectionManager = new CollectionManager(  
  this.config.contractAddress,  
  signer  
)
```

```
this.analytics = new NFTEAnalytics(  
  this.config.contractAddress,  
  this.provider,  
  this.config.deploymentBlock  
)
```

```
this.mintingSite = new MintingSite()
```

```
this.initialized = true  
}
```

```
async getUserDashboard(userAddress) {  
  const ownedTokens = await  
this.gallery.getCollectionWithMetadata(  
  this.config.contractAddress,  
  userAddress,  
  { limit: 100 }  
)
```

```
  const tokenRarities = []  
  for (const token of ownedTokens.tokens) {  
    if (token.metadata) {  
      const rarity =  
this.rarityCalculator.tokenRarities.get(token.tokenId)  
      tokenRarities.push({  
        ...token,  
        rarity: rarity || null  
      })  
    }  
  }  
}
```

```
return {
```

```
collection: ownedTokens.collection,  
tokens: tokenRarities,  
pagination: ownedTokens.pagination  
}  
}
```

```
async getAdminDashboard() {  
  const [contractState, holderStats, mintHistory, analytics] = await  
  Promise.all([  
    this.collectionManager.getContractState(),  
    this.collectionManager.getHolderDistribution(),  
    this.collectionManager.getMintHistory(),  
    this.analytics.generateReport()  
  ])
```

```
  return {  
    contract: contractState,  
    holders: holderStats,  
    mintHistory: mintHistory.slice(-50),  
    analytics  
  }  
}
```

```
async prepareAirdrop(addressList) {  
  const validation = await  
  this.airdropManager.validateAddresses(addressList)
```

```
  if (validation.valid.length === 0) {  
    return {  
      canProceed: false,  
      validation,  
      gasEstimate: null  
    }  
  }  
}
```

```
  const gasEstimate = await  
  this.airdropManager.estimateGas(validation.valid)
```

```
  return {  
    canProceed: true,
```



```
validation,  
gasEstimate,  
recipientCount: validation.valid.length  
}  
}
```

```
async loadCollectionRarity(metadataSource) {  
  let allMetadata
```

```
  if (Array.isArray(metadataSource)) {  
    allMetadata = metadataSource  
  } else {  
    const response = await fetch(metadataSource)  
    allMetadata = await response.json()  
  }
```

```
  this.rarityCalculator.analyzeCollection(allMetadata)  
  this.rarityCalculator.calculateAllRarities(allMetadata)
```

```
  return this.rarityCalculator.getCollectionStats()  
}  
}
```

```
const deploymentConfig = {  
  rpcUrl: 'https://eth-mainnet.g.alchemy.com/v2/YOUR_KEY',  
  contractAddress: '0x...',  
  deploymentBlock: 17000000,  
  chainId: 1  
}
```

```
async function main() {  
  const app = new NFTApplicationSuite(deploymentConfig)  
  
  const provider = new ethers.BrowserProvider(window.ethereum)  
  const signer = await provider.getSigner()
```

```
  await app.initialize(signer)
```

```
  const userAddress = await signer.getAddress()  
  const dashboard = await app.getUserDashboard(userAddress)
```

```
console.log('User Dashboard:', dashboard)

const adminDashboard = await app.getAdminDashboard()
console.log('Admin Dashboard:', adminDashboard)
}
```

The complete application suite integrates minting, gallery display, rarity analysis, airdrop management, collection administration, and analytics into a unified system. Each component can be used independently or together for comprehensive NFT project management.

This chapter provided production-ready code for building real NFT applications. The minting dApp handles wallet connection and transactions. The gallery displays collections with metadata. The rarity calculator helps collectors find valuable tokens. The collection manager gives creators administrative control. The airdrop system distributes tokens efficiently. The analytics module tracks project health.

These components represent the foundation of every successful NFT project. With this toolkit, you can build any NFT application from simple mint sites to complex marketplace integrations.